

An Event- and Repository-Based Component Framework for Workflow System Architecture

DISSERTATION

DER WIRTSCHAFTSWISSENSCHAFTLICHEN
FAKULTÄT
DER UNIVERSITÄT ZÜRICH

zur Erlangung der Würde
eines Doktors der Informatik

vorgelegt von
DIMITRIOS TOMBROS
von
Athen, Griechenland / Biasca, TI

genehmigt auf Antrag von
PROF. DR. K.R. DITTRICH
PROF. DR. M. GLINZ

November 1999

Die Wirtschaftswissenschaftliche Fakultät, Abteilung Informatik, gestattet hierdurch die Drucklegung der vorliegenden Dissertation, ohne damit zu den darin ausgesprochenen Anschauungen Stellung zu nehmen.

Zürich, den 30. Juni 1999

Der Abteilungsvorsteher: Prof. Dr. P. Stucki

Abstract

During the past decade a new class of systems has emerged, which plays an important role in the support of efficient business process implementation: workflow systems. Despite their proliferation however, workflow systems are still being developed in an ad hoc way without making use of advanced software engineering technologies such as component-based system development and reuse of architecture artifacts.

This work proposes a modern approach to workflow system construction. The approach is centered around a domain-specific software architecture metamodel (the REWORK metamodel) and a repository-based composition framework for workflow system construction out of reusable reactive components. The architecture metamodel defines the component and connector abstractions necessary for describing the static and dynamic aspects of a workflow system. The composition framework defines the lifecycle of a workflow system and supports the dynamic extension of a kernel workflow management system with application-specific elements. Appropriately, resulting systems are called REWORK systems.

An event- and repository-based style underlies the REWORK framework. Events are the only component integration mechanism used in REWORK systems. Repositories support both system development by storing artifacts which are used for workflow system development and system operation by making explicit the structure of a running REWORK system.

The iterative workflow system composition lifecycle proposed in this thesis comprises the following phases: the *architecture analysis* phase allows the identification and characterization of processing entities which participate in workflow execution; this phase is supported by a classification framework for processing entities in accordance to their integration-related properties. During the *architecture definition* phase workflow system components are defined and their behavior is tailored to specifications of workflows which are intended to be executed by the resulting system; furthermore, organizational relations and task assignment policies for these components are declaratively defined. The *implementation* phase is largely automated and consists in the instantiation of the defined components on top of an event-based operational infrastructure.

As previously mentioned the entire lifecycle is supported by repositories which store the workflow system artifacts. The reuse-based development process comes into the picture once existing workflow systems have to be maintained either by adding new repository artifacts or by modifying existing ones to suit new requirements. Thus, we dedicate a part of this thesis to the description of these repositories.

Zusammenfassung

In den letzten Jahren ist eine neue Klasse von Systemen entstanden, die eine bedeutende Rolle bei der effizienten Realisierung von Geschäftsprozessen spielen: Workflow-Systeme. Trotz ihrer Ausbreitung werden Workflow-Systeme immer noch ad hoc entwickelt, ohne fortgeschrittene Software Engineering Techniken, wie z.B. komponentenbasierte Systementwicklung und Wiederverwendung von Architekturartefakten einzusetzen.

Die vorliegende Arbeit führt einen neuen Ansatz für die Konstruktion von Workflow-Systemen ein. Der Ansatz basiert auf einem domänenspezifischen Architektur-Metamodell (dem REWORK-Metamodell) und einem repository-basierten Rahmengerüst (dem REWORK-Framework) für das Zusammensetzen von Workflow-Systemen aus wiederverwendbaren, reaktiven Komponenten. Das REWORK-Metamodell definiert die verfügbaren Komponenten- und Verbindungselement-Abstraktionen, welche für die Beschreibung der statischen und dynamischen Aspekte eines Workflow-Systems notwendig sind. Das REWORK-Framework definiert einen Lebenszyklus für die Implementierung und nachträgliche Erweiterung von Workflow-Systemen durch anwendungsspezifische Elemente. Die resultierenden Systeme heissen REWORK-Systeme.

Ein ereignis- und repository-basierter Stil charakterisiert das REWORK-Framework. Ereignisse stellen den einzigen Integrationsmechanismus in REWORK-Systemen dar. Die Entwicklung und der Betrieb von REWORK-Systemen wird durch Repositories unterstützt, welche Entwicklungsartefakte und die Laufzeitarchitektur von REWORK-Systemen explizit verwalten. Der iterative Workflow-Konstruktionslebenszyklus, welcher im Rahmen dieser Arbeit vorgestellt wird, beinhaltet folgende Einzelphasen: Die *Architekturanalysephase* ermöglicht die Identifizierung und Charakterisierung der Verarbeitungseinheiten, welche an der Workflow-Ausführung beteiligt sind. Diese Phase wird durch einen Klassifikationsansatz und konzeptuelle Werkzeuge unterstützt, die auf die Integrationseigenschaften ausgerichtet sind. Während der *Architekturdefinitionsphase* werden Workflow-Systemkomponenten spezifiziert und ihr Verhalten wird entsprechend der vorliegenden Workflow-Prozessspezifikationen definiert. Schliesslich werden organisatorische Beziehungen und Aufgabenzuweisungsstrategien festgelegt. Die *Implementierungsphase* ist weitgehend automatisiert und besteht aus der Instantiierung der definierten Komponenten auf einer geeigneten ereignisbasierten Ausführungsplattform.

Wie bereits erwähnt wird der gesamte Lebenszyklus durch Repositories unterstützt, die die Workflow-Systemartefakte speichern und verwalten. Der iterative Entwicklungsprozess kann dann durch Hinzufügen von neuen oder Anpassen von bestehenden Repository-Artefakten realisiert werden. Ein Teil dieser Arbeit ist daher der Beschreibung dieser Repositories gewidmet.

Acknowledgements

This thesis has been prepared during my employment as a research assistant at the Department of Information Technology of the University of Zurich. I thank the Swiss National Fund and the Ministry of Education of the Kanton of Zurich for funding my position.

I would like to thank my advisors. Prof. Klaus R. Dittrich has provided me with guidance and inspiration during these past years. I gratefully acknowledge his support and encouragement during the preparation of this thesis, as well as his helpful comments during the review of my work. He has also greatly contributed to a challenging but nevertheless friendly working environment. I would also like to thank Prof. Martin Glinz who agreed to provide a review and contributed many valuable comments.

I gratefully acknowledge the contributions of my colleagues in the Database Technology Research Group at the Department of Information Technology. Most of them have contributed to this work in one way or another. I especially thank Dr. Andreas Geppert for his support in our joint work and for the extensive discussions which served in illuminating the long and sometimes mazed path towards the completion of this thesis. I also would like to thank Markus Kradolfer for sharing his deep insight in workflow modeling issues, Dr. Stella Gatzia for her support in clarifying issues related to active database systems, and Dr. Dirk Jonscher for often assisting me in finding the best terminology. Finally, I thank all the anonymous reviewers of my submissions to scientific conferences which often contributed valuable comments.

My parents, Aliko and Marios Tombros, have not only funded a great deal of my education, they have also taught me the value of education. I would like to express my deep gratitude for always supporting me during the progress of my studies and being sure about the success of my effort!

Last but not least, I would like to thank my wife Meri. Without her trustful, patient, and loving support it would have been much harder to accomplish this work. Therefore this work is dedicated to her.

Contents

1	Introduction and Problem Statement	11
1.1	Workflow Management Technology	11
1.2	Motivation	13
1.3	Outline of the Solution	15
1.4	Outline of the Thesis	19

Part I: Workflow Management

2	Technological Background	23
2.1	Software Architecture	23
2.2	Database Systems	27
2.3	Information System Integration	32
2.4	Event-Based Integration	37
2.5	Summary	45
3	The Workflow System Domain	47
3.1	The Business Process Perspective	47
3.2	From Business Processes to Workflow Systems	48
3.3	Workflow Specification	50
3.4	Architecture Styles for Workflow Management Systems	56
3.5	Workflow Application Metamodels	60
3.6	Summary	63

Part II: Workflow System Architecture

4	Analysis of Workflow System Integration Architecture	67
4.1	From Actors to Integration Components	67
4.2	Examples of Workflow Application Systems	69
4.3	Implementation Architecture of WFMS	73
4.4	A Classification Scheme for Workflow System Actors	82
4.5	Summary	91
5	Events in Workflow Systems	93
5.1	Motivating an Event-Based Workflow System Metamodel	94
5.2	Services and Workflows	97
5.3	Primitive Events in the Workflow System Metamodel	100
5.4	Composite Events in the REWORK Metamodel	106
5.5	Event History in a REWORK System	114
5.6	Summary	115
6	Reactive Workflow System Components	117
6.1	Event Occurrence Brokers	117
6.2	Actor Representations: The REWORK Components	118

6.3	Organizational Relationships	130
6.4	Task Assignment in REWORK Systems	132
6.5	Event History in a REWORK System	135
6.6	Summary	138
7	Workflow Specification	139
7.1	Integrating Workflow Specifications in REWORK Architectures	140
7.2	Mapping the WfMC Process Metamodel to the REWORK Metamodel	142
7.3	Market-Based Task Assignment	150
7.4	Summary	153
 Part III: Workflow System Composition		
8	The REWORK Build-Time Repository	157
8.1	Repository-Based Component Reuse	157
8.2	A Build-Time Repository for the REWORK Metamodel	158
8.3	Summary	163
9	The REWORK Run-Time Architecture	165
9.1	EOB Implementation	166
9.2	Messaging	167
9.3	Distributed Event Detection	168
9.4	Run-Time Repository Organization	172
9.5	Event History Management	173
9.6	Summary	173
10	The Workflow System Lifecycle Revisited	175
10.1	Avoiding Architectural Mismatch with REWORK	175
10.2	The REWORK System Development Lifecycle	177
10.3	Summary	180
11	Conclusion	181
11.1	Contributions	181
11.2	Directions for Future Work	182
List of Abbreviations		185
Glossary		187
References		191

1 Introduction and Problem Statement

The period of world-wide recession in the early 1990's and the increasing competitiveness in world markets has motivated corporations to reexamine and streamline existing organizational structures as well as ways of accomplishing business tasks and reaching their business goals. Principal vehicles of these changes are the notions of *lean management* and *business process re-engineering* [Hammer & Champy, 1994]. One of the main results of the application of these concepts has been the increased emphasis on the optimization of business processes and the analysis of the critical factors for their success.

The effect of this cultural change on information technology has been important. *Workflow management (WM)* has become an active research topic in applied computer science during the past decade, as well as one of the recent buzzwords in IT-literature. Workflow management is generally defined as the principal supporting technology for the automation of business processes [Georgakopoulos et al., 1995]. It includes the description of the aspects of a business process that are relevant for the control and the coordination of the execution of its constituent tasks and the provision of technologies for the implementation of the process.

1.1 Workflow Management Technology

The term “workflow” is defined by the Workflow Management Coalition (WfMC)¹ as the computerized facilitation or automation of a business process, in whole or part. A *workflow management system (WFMS)* is a system that completely defines, manages, and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic [WfMC, 1994]. In other words, workflows are formally described business processes whose automatic execution is supported by workflow management systems. A *workflow* is a collection of tasks which are performed by software systems, people and groups of people, or a combination of both [Georgakopoulos et al., 1995]. A *workflow system* consists of a WFMS and the integrated *processing entities* or *actors* composing the *workflow application system* (or simply *workflow application*) which are responsible for executing workflow tasks. The workflow system actors can be automated software entities or human beings.

Workflow management is the enabling technology for automating (parts of) business processes through the connection of individual tasks in a value chain². Due to the interactive nature of many workflow tasks, only partial automation can be achieved or may even be desirable, especially if the anthropocentric view of modern business practices is considered. Even automated tasks may be initiated by people who may subsequently decide whether such tasks have successfully completed or not.

¹ The WfMC is a consortium of vendors which provide workflow management-related products.

² The term “workflow automation”, often used in connection with the automatization of business processes, is considered by some authors somewhat differently as it aims more towards the automated execution of interdependent applications [Rusinkiewicz & Sheth, 1995], i.e., it refers to application coordination. We do not make a distinction in this thesis.

Frequently stated business objectives and expected advantages of the use of WM technology include the following:

- It serves in improving the performance of an organization by (partially) automating processes. Furthermore, it has the potential to facilitate the implementation of process and policy changes.
- It improves the business competitiveness and productivity of an organization. It provides just-in-time information and has the potential of improving timeliness of product delivery. In addition, it provides an instrument for process quality control.
- It provides a framework for an organization to manage its process chains (to understand them, analyze their performance and deficiencies, and eventually optimize them). Thus, workflow technology effectively implements a continuous business process optimization loop.
- It provides an integration platform for existing but isolated information systems. Intra- and potentially inter-organizational information exchange can be facilitated and information consistency regarding business transactions can be achieved.

Organizations usually introduce workflow management technology as a result of business process redesign and automation. Most organizations start with pilot projects in which the technology is assessed and know-how is gradually accumulated. Once basic familiarity with the technology is acquired, further workflow application systems are implemented. It is often desirable that the islands of automation are connected to form enterprise-wide workflow systems or even systems which cross organizational boundaries. This however is still not feasible with current WM technology as is obvious from recent research in this direction, e.g., [Alonso & Schek, 1996, Riempp & Nastansky, 1997].

Due to the business-driven development of workflow management, and contrary to the case of technologies such as relational database systems or object-oriented programming which have been implemented following the development of sound theoretical foundations, the initial emphasis in workflow management has been in providing the market as quickly as possible with operational products. This becomes obvious when we consider the plethora of workflow management-related products which entered —and often subsequently left (!)— the market during the last years. At the end of 1996, for example, more than 100 vendors of workflow management products were listed in the database of the Workflow and Reengineering International Association (WARIA). Recently however, a trend towards market consolidation and concentration on specific products can be observed.

Workflow management in general and workflow systems in particular have their historical origin in a variety of technologies such as office automation systems and document management systems. Furthermore, various technologies such as software process management, business process modeling, and enterprise modeling, active database systems, and advanced transaction models have influenced the development of WM technology in some important way [Jablonski & Bussler, 1996]. It is thus obvious that WM is a multi-disciplinary area of information technology. A related perspective is provided by [Hsu & Kleissner, 1996] who distinguish between the following phases of historical development of workflow systems:

- *First generation: homegrown workflow systems.* This phase which lasted until about 1992 is characterized by monolithic architectures and the hard-coding of information and data flow into the applications.

- *Second generation: rudimentary workflow.* This generation which lasted from 1992 to 1995 was driven by imaging and document management systems (DMS), desktop object management systems, or business-oriented modular applications. This fact still influences many products in the market which have evolved as enhancements of well established DMS applications. Thus the workflow functionality is typically provided by a component which cannot be separated from the rest of the system.
- *Third generation: “architected” workflow.* This generation of systems which started at about 1994 is characterized by generic workflow engines with open interfaces which provide the infrastructure for production-oriented or administrative workflows and in some cases even ad hoc workflows. These systems use various middleware technologies for information sharing, communication, and distribution. Examples of such systems include most current research prototypes which will be considered in this thesis.

Similarly, [Abbot & Sarin, 1994] distinguish between an experimental phase (– 1993) whose outcome was a working knowledge of modeling and execution issues, a conceptual phase (1992 – 1998) in which models and architectures were developed, and a standardization phase (after 1994). From our vantage point of 1998, we believe that the conceptual phase has not yet been completed.

Despite the large body of research that has been performed in WFMS during the various mentioned phases and the large number of commercially available systems, it is recognized that there are still a lot of open issues and limitations in existing WM technology. These limitations, which affect the practical use of this technology, have been discussed by the workflow research community, e.g., [Alonso & Schek, 1996, Kamath & Ramamrithan, 1996, Tombros & Geppert, 1997]. This thesis attempts to conceptualize and resolve some of these still open issues.

1.2 Motivation

Traditional software development methods do not provide adequate support for the evolving requirements of large scale heterogeneous and process-oriented systems. A principal open problem remains the systematic development of workflow systems based on appropriate abstractions and mechanisms. In this thesis, we propose a solution to this problem by advocating an architectural view of workflow systems as flexible compositions of actors implemented by reactive software components. This view is supported by appropriate conceptual tools, a composition method, and implementation support in the form of an architectural framework for workflow systems.

The current state of workflow system research and technology has not yet reached consensus on the proper software engineering abstractions for the workflow domain and on the development process and life cycle of workflow application systems. Instead, there is a proliferation of research proposing the benefits of particular workflow management concepts, each emphasizing some aspect of workflow systems that the authors deem most important. There are some recent efforts [Weske et al., 1999] that attempt to converge various ideas in the workflow management community. However, workflow system development remains a largely *ad hoc* effort done on a case-by-case basis.

We believe one of the fundamental things missing in current workflow system research is a proper foundation, or architectural framework, which can be used for the systematic development of workflow systems. Such a framework should address workflow system development and workflow artifact reuse issues and representations at an appropriate level of abstraction. It should also provide for extensions to the available abstractions where necessary. The WfMC has attempted to

do this in the context of their Reference Model [WfMC, 1994] and corresponding workflow application programming interfaces (API) and process interchange (WAPI) interfaces [WfMC, 1995, WfMC, 1996a, WfMC, 1996b, WfMC, 1996d]. While the Reference Model provides a high-level discussion framework for workflow system organization, it does not do as well with respect to abstractions of process definition and actor representations. As a result, we feel that using the respective WAPI for manipulations of the workflow domain, especially for providing extensibility of workflow systems and for reuse of workflow system artifacts, is insufficient from a software engineering perspective. It is our opinion that better abstractions are required for a suitable architectural framework.

Workflow application systems have various characteristics which make their design and construction extremely demanding. Their primary purpose when abstracting from the concrete business processes is the support for the integration and interoperability among *distributed*, *heterogeneous*, and *autonomous* legacy and new application software:

- The participating systems typically operate in a *distributed environment* consisting of various computing sites connected by communication networks of various topologies and technologies.
- The participating software systems are *heterogeneous* with respect to their implementation technologies, their underlying operating platforms, and their application domains. This heterogeneity has an important impact on the provided interfaces, their interaction models, and other underlying assumptions.
- The participating software systems have often been conceived as *autonomous* software entities in the sense that they are able to provide business functionality independently or in cooperation with other systems, without necessarily being part of a workflow application system.
- The participating software systems often need to be *customized* to meet the requirements of specific workflow applications. The necessary customizing must be *localized* and may not affect other parts of the system.

On the other hand, various requirements have to be satisfied by the entire workflow system. A WS must be reliable and execute workflows in a correct way even in the presence of failures and concurrency. Furthermore, they must be able to evolve over time through the addition, deletion, and modification of participating actors required both by changing technological and business-related requirements. Despite the recent emergence of interoperability standards (e.g., the CORBA architecture [OMG, 1995]), the development of distributed, process-oriented information systems poses complex problems which are currently the subject of intensive research [Papazoglou & Schlageter, 1998]. There is however consensus that a *composition-based development* of cooperative information systems based on *reusable components* provides the only viable approach.

The component-based software development approach underlies currently developed component technologies such as Microsoft's ActiveX/DCOM [Dennings, 1997] or Sun's JavaBeans [Sun Microsystems, 1997]. Such generic domain-independent component technologies still lie however at a rather low level of abstraction, provide limited support for composition and location transparency, and have to be enriched with a large body of domain-specific semantics and functionality. We thus believe that application-domain-oriented high-level software components —as proposed in this thesis— which provide rich domain-specific functionality and therefore limit the required customizing can more efficiently facilitate the development of workflow systems.

1.3 Outline of the Solution

The classic approach to workflow application development involves a phased approach (depicted in Figure 1-1 (a).) whose products are a conceptual understanding of the business process, a workflow specification, and the implemented workflow application system [Georgakopoulos et al., 1995]. The *workflow specification* captures a process abstraction using concepts provided by a workflow metamodel which can be described in a formal workflow specification language. The *workflow implementation* is based on an implementation method based on the specific mechanisms provided by the chosen WFMS. The result of the implementation is an operating workflow application system which is able to support the execution of the specified business process. In various systems, no clear dividing line can be drawn between workflow specification and workflow implementation. In these cases, the specification and implementation mechanisms are tightly coupled. Furthermore, the functionality provided by the WFMS cannot be extended to support special needs of particular workflow applications.

This classic workflow system development approach has important deficiencies which preclude the efficient composition-based implementation of systems with the required properties and does not effectively support reuse. We believe that the introduction of an intermediate layer of abstraction between the business-process oriented workflow specification and the physical system implementation is required. This intermediate layer provides the following important advantages to workflow system development:

- The introduction of a logical view of the workflow system architecture allows the systematic composition of workflow systems and facilitates abstract reasoning about their properties.

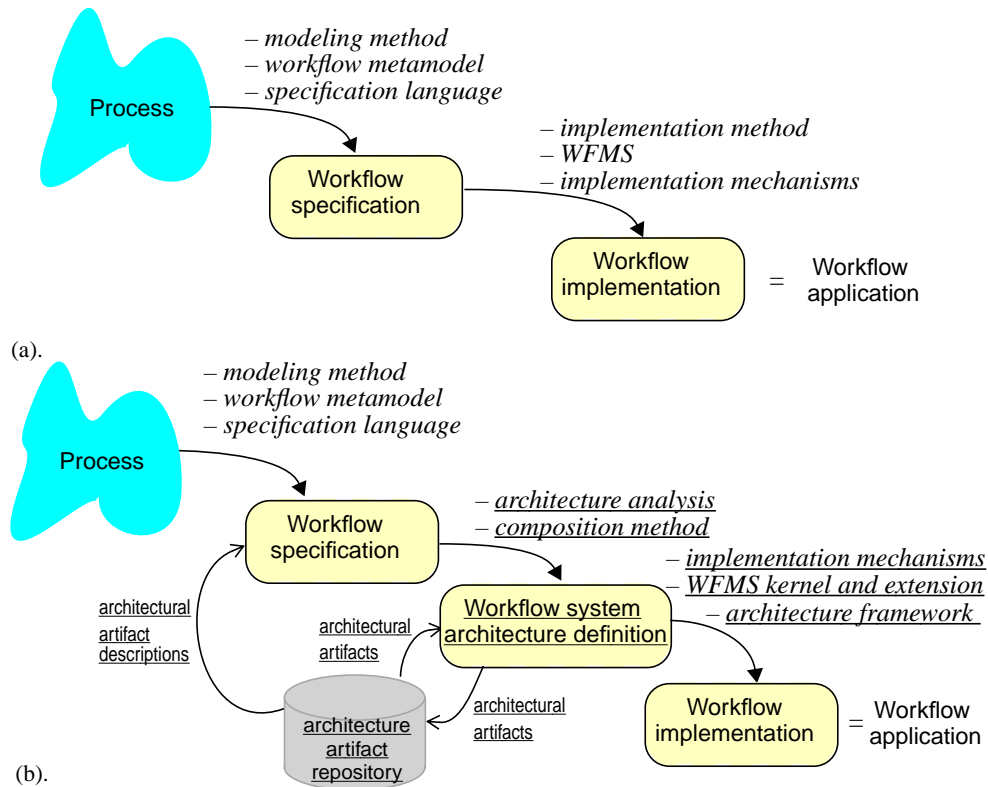


Figure 1-1: Workflow application development process (a). traditional and (b). composition-based. The underlined elements of the diagram refer to contributions of the thesis.

- A conceptual framework is defined in which different workflow specification formalisms can be mapped to a uniform system model with defined formal semantics. This is especially important, as it allows the use of special-purpose optimized specification languages to describe different aspects of a workflow system or different workflow subsystems, which are subsequently integrated in a uniform architectural model.
- The link between workflow specification and workflow system architecture is provided by an architecture artifact repository in which workflow system components are stored and maintained. These components can be used to compose a workflow system, taking into consideration the analysis of the specific workflow application. The reuse-based development of workflow systems is promoted.
- The workflow implementation is based on the description of the system architecture; this can be transformed in a largely automated way to an operative system through implementation mechanisms provided by the architecture framework.

The new development process is depicted in Figure 1-1 (b). It is a well-known fact that a layered approach can contribute to the reduction of complexity in engineering problems. In our work we propose a layered conceptual architecture for workflow systems:

- *Application and process definition layer.* The topmost layer is the domain of workflow modeling in which declarative specifications of the executing workflows are created. The representational constructs of this layer are provided by workflow specification languages. These declarative workflow specifications take into consideration abstract descriptions of the underlying physical system.
- *Architectural layer.* The middle layer describes the software architecture of the workflow system composed of predefined and additional customized components provided by an architectural framework. The representational constructs of this layer are reusable reactive software components representing the workflow actors and the workflow system infrastructure, connectors allowing event-based interaction, and abstractions for the definition of organizational relationships and task assignment policies. Furthermore, the architectural view provides constructs to express and control the formal semantics of workflow execution in a given workflow system.
- *Physical layer.* The bottom layer describes the workflow execution and integration infrastructure. The representational elements of this layer are implementation-level constructs.

In this thesis we concentrate on the architectural view of workflow systems. This is proposed by an architecture-oriented workflow system development environment called the *REWORK environment*. It includes the *REWORK metamodel* providing representation mechanisms and concepts for the description of reusable reactive components, i.e., architecture-level artifacts, used for the composition of workflow systems. The components are integrated by an event-based architectural framework which provides mechanisms to express formal workflow execution semantics. REWORK also supports a composition method for workflow system development. We consider the interfaces of the architectural layer to the layers above and below it by providing mapping mechanisms from workflow specifications to REWORK architecture models and from those to an appropriate physical layer. The various aspects of the REWORK environment are considered in more detail in the next subsections.

1.3.1 Event-Based Workflow System Architecture

The architecture of a system has important repercussions on both its functional and non-functional properties. While initially workflow systems were built in a monolithic fashion, the requirements of large-scale distributed workflow execution have been considered by more recent approaches which propose multi-site, multi-server architectures and are often constructed on top of a distributed computing infrastructure (e.g., CORBA [OMG, 1995]). While this infrastructure does resolve distribution and interoperability issues—at least for the integration of systems conforming to the standards—the flexibility and evolving capacity of the system is still dependent on the functionality provided by the underlying implementation platform, the dependencies hard-coded in participating components, and the implicit assumptions that must be made during workflow specification about the functionality provided by the WFMS. This is usually manifested when modifications or extensions are made, which result in changes required in various parts of the system, a well known problem in large systems.

Through the introduction of the conceptual view of the workflow system, the dependency between specification and implementation is made explicit. The proposed metamodel for the conceptual architecture is event-based. The actors are represented by software components which generate events and define reactions to events occurring in their environment. The resulting reactions describe the processing that takes place in the component when an event generated by other such components occurs, effectively resulting in the desired component interactions. Thus, a novel event-based workflow execution paradigm is introduced in which the state of workflow execution is externalized by the occurrence of events in the workflow system.

The event-based architectural style provides the well-known advantages of implicit invocation systems [Garlan & Notkin, 1991], specifically facilitating the composition of open, flexible and extensible systems in which the participating components make no assumptions about other participants. This is in contrast to message-based approaches (e.g., CORBA [OMG, 1995]) used in most modern workflow systems, in which the recipient of a message has to be determined before the message is sent, thus requiring additional layers of functionality to support more sophisticated caller/recipient interactions such as multiple or alternative recipients.

In addition to the integration of workflow applications, administration facilities like worklist management, monitoring, and logging are also needed in most workflow application systems. The problem here is that the requirements different workflow applications pose on these facilities are as diverse as the workflow applications themselves. For example, the need for complex strategies for assigning tasks to human actors, such as load balancing between actors, cannot be predicted in advance. These principles have to be implemented on demand and have to be tailored to specific needs of a workflow application system. Various system services might be required. This observation is not properly reflected in most architectures of workflow management systems today, as administration components are not described explicitly. As a consequence, these components cannot exactly match the requirements of different workflow application systems. They are either not powerful enough and can not be sufficiently tailored to application needs, or they provide too much functionality which, in most cases, remains unexploited but increases the system cost, the system footprint, and affects the system performance. The proposed event-based architecture supports the extensibility of the basic WFMS functionality by allowing the customizing of existing and addition of new components, keeping however the changes localized to the affected components. Components interact through asynchronous event broadcasting and thus need not be aware of the location and interfaces of other components to use their functionality.

While event-based systems have been proposed as integration platforms by various researchers—especially in the software engineering community—many issues pertaining to event-based

distributed workflow execution have not yet been resolved. In this thesis, we consider these open issues and propose a solution comprising an event-based workflow system architecture and a supporting event-based implementation infrastructure.

Summarizing, the first contribution of this thesis is the *application and adaptation of event-based interaction in workflow systems* through an appropriate event-based architectural framework.

1.3.2 Semantics of Workflow Execution

In order to reason about correctness properties of workflows executed in a given workflow system, the formal semantics of these workflows must be definable. This requirement has been recognized by some of the most recent research in the field (e.g., [Wodtke, 1997]) and is especially true when distributed workflow execution takes place. The existence of formal semantics for a workflow modeling formalism allows the definition and the proof of correctness properties of these workflows.

While many workflow specification languages are based on representations with formally defined semantics, the implementation of the workflow application systems often relies on ad hoc mechanisms. In this thesis, we propose an event-based workflow specification approach which is directly mapped to an event-based workflow execution infrastructure. The formal semantics of the resulting workflow execution are expressed by an event algebra. This permits the analysis of the correctness workflow execution with respect to workflow specification.

Summarizing, the second contribution of this thesis is the development of *clearly defined operational semantics* for workflow execution onto which a large class of workflow specification languages can be mapped. Support for the mapping is provided by the REWORK environment.

1.3.3 Composition of Workflow Systems

Component-based workflow system composition is still in a very early stage. This is not only an inherent problem of workflow management technology but can also be explained in the general context of system composition research. In our work we considered the following facts as especially relevant for workflow system development:

- It is not clear how domain knowledge should be captured and formalized to support component-oriented development. Especially in the domain of workflow management, there has been little or no research on this issue. The definition of complete development and reuse-oriented domain-specific reference models is still an ongoing process.
- The relation between workflow specification and workflow implementation has not been considered under the perspective of reusability. For example, most workflow specification languages provide reuse mechanisms for workflow artifacts but do not consider the reuse of actor implementations. The mapping of workflow specifications to workflow system implementation is ad hoc.
- There are no generally accepted methods for the design of frameworks supporting component-based workflow system development. Object-oriented analysis and design methods do not address the development of frameworks. Especially in the domain of workflow management, an architectural perspective and the resulting framework-based development approach is in a very early stage.

- There are no or only limited software tools and environments which facilitate component-oriented workflow system development. The support provided by workflow management systems in this respect is limited to workflow specification. These specifications then are executed by a monolithic workflow management system.
- A difficulty inherent in the composition of large complex systems, known under the term '*architectural mismatch*', has been recognized by the software architecture community [Garlan et al., 1995]. It occurs when independently developed heterogeneous component systems, which make conflicting assumptions about the composed system, have to be integrated into a global architecture. These assumptions refer to the nature of other components, the nature of the interaction mechanisms, and the construction process of the composed system.

Only recently has there been a change of perspective towards an architectural consideration of workflow systems. Often this is a by-product of the use of a particular implementation technique; for example, components are introduced by the use of a CORBA-based communication infrastructure in a workflow system. As already discussed, the description of system architecture in most current workflow systems is ad hoc and is dispersed in both the workflow specification as well as the workflow system implementation level.

The composition-based approach for workflow system development proposed in the REWORK environment supports the architecture-centric development and extension of workflow systems. It consists of the following elements:

- A framework for the *analysis* of the workflow system architecture and the *classification* of workflow system components.
- A domain-specific metamodel, called the *REWORK metamodel*, for the specification of the *architecture* and *functionality* of the intended workflow system.
- A framework for the *composition* of workflow systems out of parameterized component templates which try to solve various issues related to architectural mismatch. The added advantage is that the required workflow management infrastructure can be optimized towards the requirements of specific workflow application systems by extending the lightweight kernel WFMS through additional components.
- Software support for the composition of workflow systems through an *architecture artifact repository* and a *generic event-based execution platform*. This infrastructure provides a *workflow specification execution system*.

Thus, the third contribution of this thesis is a *composition-based approach* for the development of *extensible workflow system architectures*. The architectural mismatch problem is alleviated by the mapping to a uniform component metamodel. Conflicting assumptions made by the integrated subsystems are encapsulated in REWORK components.

1.4 Outline of the Thesis

This thesis is structured as follows: part I introduces the relevant technology and application domain. In chapter 2 we describe the relevant technological background for the proposed solution. We discuss the nature of software architecture, introduce active database technology, and middleware with particular emphasis on event-based integration mechanisms. In chapter 3 we analyze the application domain of workflow systems. We consider the transition from business processes to workflow specifications and survey various specification approaches.

Part II forms the core of this thesis. In chapter 4 we propose an analysis approach for workflow system architectures. We survey workflow application systems and WFMS from an architectural perspective and subsequently define a characterization framework. In chapters 5 and 6 we describe the elements and operational semantics of an event-based architectural metamodel for workflow systems. In chapter 7 we describe the use of the metamodel for the transformation of existing workflow specifications to executable workflow systems.

Finally, part III concludes this thesis. In chapter 8 we consider the implementation of an environment for workflow system composition and a repository supporting the reuse of workflow system architectural artifacts. In chapter 9 we describe the implementation of a distributed event engine for workflow system execution. In chapter 10 we present in some detail the life cycle for REWORK-based workflow systems. We summarize our work and conclude with a discussion of related open issues in chapter 11.

Part I: Workflow Management

In the first part of this thesis, which includes chapters 2 and 3, we focus our discussion on the domain of discourse. As mentioned in the introduction, workflow management is a multi-disciplinary research area, in which technologies from various domains have contributed to its development.

In chapter 2 we describe the technologies which are most relevant to the development of the solution proposed in this thesis. In particular, we discuss the field of software architecture to provide a conceptual framework for our work. Furthermore, we discuss object-oriented database technology, as it influences in an important way both the development and the operation of workflow systems. In our work we apply database technology both for the build-time environment, which is based on a software artifact repository, and the run-time environment, which again is based on an appropriate repository of run-time components. We focus on active database technology as a non-standard database paradigm from which many elements of this work have been derived. We conclude the chapter by discussing middleware technology in general, concentrating particularly on the technology of event-based coordination and system integration.

In chapter 3 we turn our attention to the domain of workflow systems. We specifically survey the various workflow specification approaches, as well as the mechanisms provided for application integration. We evaluate these mechanisms as foundations for operational workflow system architectures.

2 Technological Background

As mentioned in the introduction, various domains have influenced the development of workflow management technology. In this chapter we consider these technologies which provide the foundations for the concepts and mechanisms developed in this thesis. In the first section we place our work in the context of software architecture research. Subsequently, we elaborate on two domains of information technology which provide the technical background for various concepts developed here. In particular, we consider the following subjects:

- database technology in general emphasizing on active database systems; and
- middleware in general and event-based integration frameworks in particular.

2.1 Software Architecture

Software architecture is an emerging field of research in software engineering. It has evolved from low level abstractions used for the representation of software system structure. Software architecture considers a software system from the perspective of global organization as a composition of components, global control structures, communication protocols, and physical locations over which the system is distributed. It also considers design issues such as assignment of functionality to design elements, dimensions of system evolution, and selection among design alternatives. Thus, the scope of software architecture is the description of composition elements of a system, their interactions, composition patterns and constraints on these patterns [Shaw & Garlan, 1996]. In this section, we describe notions and concepts of software architecture and architectural style which are relevant to our work.

2.1.1 Advantages of Architecture Descriptions

Architectural elements play an important role in the description of complex systems. The formal definition of architectural elements and software architectures is a requirement for its use in the ways described below [Abowd et al., 1995, Garlan et al., 1995, Bass et al., 1998]:

- The communication between end-users, software developers, and management—especially important when developing large complex systems—is facilitated when these systems can be discussed at a sufficiently high level of abstraction. Architecture provides a common language in which the perspectives and interests of the different stakeholders can be expressed, negotiated, and resolved.
- It is possible to define constraints on the structure and topology of complex systems independently from the implementation of component systems. The software system then must comprise the prescribed components, which must interact with each other in the predefined ways and must relate to other components in the prescribed fashion.
- Large systems can be built through methodical composition and reuse of architectural elements. Design at the architectural level supports the reuse of large components in contrast

to reuse of fine granularity elements through class libraries. The emphasis thus shifts from software development to software composition. Additional components can be incorporated into the system or existing components can be replaced by new ones if they obey the constraints defined by the architecture. There are still however unresolved issues concerning architectural composition, a problem termed “architectural mismatch” [Garlan et al., 1995].

- The implementation of an *architectural framework* for a specific class of software systems allows the subsequent rapid production of new systems by template instantiating. This is especially so if domain-specific architectures are defined in which fundamental design decisions concerning allowed component types and interactions are part of the architecture.
- The effects of changes of individual components on the structure and functionality of the system can be more effectively identified and isolated. From an architectural perspective, changes can be classified as local to a single component, non-local affecting multiple components but leaving their interaction patterns intact, and global affecting the entire system architecture. System evolution is more effectively supported if the impact of changes can be systematically analyzed.
- The software architect can analyze and formally reason about correctness of systems and system behavior.
- The performance of a system can often be predicted and analyzed based on architectural descriptions. In distributed systems, run-time aspects are often related to the size and complexity of inter-component interaction, as well as on the number of different software layers composing the system.

In order to achieve the aforementioned advantages, the abstractions used for the specification of system architecture must have certain properties which support system composition and configuration, as well as formal analysis and reuse of the descriptions. These properties depend on the intended purpose of the specification mechanisms.

2.1.2 Components, Connectors and Architectural Styles

In this thesis we consider the software architecture of workflow systems. We use the following definition of software architecture from [Shaw & Garlan, 1996]:

Definition 2-1: (Software architecture)

The *software architecture* of a system is defined as a collection of computational *components* and the descriptions of the interactions between these components—the *connectors*.

Thus, the basic concepts of a software architecture are computational components—or simply components—and connectors. Depending on the perspective of the architectural description, the concepts behind these notions may be somewhat different. Generally, architectural descriptions of complex systems have to deal with the individual components of these systems, the component connectors or interfaces, and the relationships between the access points of the component’s interface.

An in-depth discussion of connectors in software architecture can be found, for example, in [Allen & Garlan, 1997]. With respect to component-based technology, [Szyperski, 1997] defines a system architecture as consisting of a set of component platform decisions, a set of component

frameworks, and an interoperability design for component frameworks. In our work we adhere to his component framework definition (our additions in brackets):

Definition 2-2: (Component framework)

A *component framework* is a dedicated and focused [software] architecture, usually [based] around a few key mechanisms, and [defining] a fixed set of policies for mechanisms at the component level.

The component framework usually contains reusable chunks of domain expertise organized along various dimensions. The present thesis describes a component framework for workflow systems, called the REWORK framework. The following definitions are based on an emerging consensus of terminology in software architecture (e.g., [Abowd et al., 1995]) and are provided here for the sake of clarification of the scope of our work:

Definition 2-3: (Computational component)

A computational component describes a localized independent computation. It consists of a *computational interface*—or simply interface—and a *computational content*—simply content.

The computational interface describes two aspects of the component: it describes some of its behavior and it defines an expectation that the component has with respect to its environment. Access points generalize the notion of a module interface. They can signify anything, from a procedure that can be called, to a database access protocol.

Definition 2-4: (Computational interface)

A computational interface consists of a set of *access points* or *ports* which represent interactions in which the component may participate.

The complete component specification is available once the component's *computational content* is given. This can have the form of a set of classes and possibly non-object-oriented constructs [Szyperski, 1997]. Note that during the component construction process, components are not necessarily considered as black box entities. This is in contrast to component-based system composition in which the interns of a component are not known to its users. Note also, that two components may have the same access points but different computational contents.

Definition 2-5: (Computational content)

The computational content of a component relates the behavior of the access points of the component's interface.

Two important properties of components are that they are independently deployed and represent units of versioning and replacement [Szyperski, 1997]. In this thesis we adhere to the notion of a component being a template for building *component instances* with a unique identity.

Software architectures can be characterized, on a general level, as pertaining to a specific style [Abowd et al., 1995]. An *architectural style* defines a family of systems in terms of a pattern of structural organization. It defines a common vocabulary of component and connector types, and a set of constraints on how these can be combined. Styles which are often mentioned in the literature include pipes-and-filters and their specializations such as pipelines or batch sequential systems (e.g., the UNIX pipe mechanism), event-based or implicit invocation systems (as described below), and repository systems built around a common data store. A style is characterized

by certain invariants and systems that follow this style have certain types of components and connectors. The components of object-oriented systems, for example, are objects managing their private resources; the connectors are the messages these objects understand. In repository-based architectures, the components are the central data store and the programs operating on it; the connectors are the database access protocols.

An architectural style can be pragmatically described by an *architectural pattern* [Buschmann et al., 1996]. An architectural pattern is a template for a concrete software architecture which expresses a fundamental structural organization schema for an entire software system. It provides predefined subsystems, their responsibilities, and rules for the relationships between them. In comparison, design patterns [Gamma et al., 1995] are smaller in scale and tend to influence primarily the architecture of subsystems. In general, an architectural style can be characterized by answering the following questions:

- what is the design vocabulary—the types of components and connectors?
- what are the allowable structural patterns?
- what is the underlying computational model?
- what are the essential invariants?
- what are the advantages and disadvantages of the style?

At this point and in the interest of positioning our work, we make a leap forward and characterize the architectural style of the REWORK component framework proposed in this thesis for workflow systems. It is essentially based on a combination of the following ‘pure’ architectural styles: *event-based*, *repository-based*, and *client/server*.

2.1.3 Domain-Specific Software Architectures

Domain-specific software architectures have been proposed by various researchers as well as in software intensive businesses. They are architectures useful to a specific well-defined application domain. Examples include avionics, graphical user interfaces (GUI) [Taylor et al., 1996], and intelligent agents (e.g., [Hayes-Roth et al., 1995]). In general, a domain-specific architecture may comprise one or more of the following elements:

- a *reference architecture* which describes a general conceptual framework for applications in the domain. A reference architecture is based on a *reference model* which divides the functionality in a given domain and defines the data flow between the pieces. In the reference architecture, this model is mapped onto a system decomposition. i.e., software components that implement this functionality [Bass et al., 1998].
- a component framework (see Definition 2-2) specific to the domain; and
- an application composition method for selecting and configuring components within the architecture to meet particular application requirements.

A domain-specific architecture can be supported by an architecture development environment which allows the definition or selection of individual components and the (partially) automated generation of executable systems [Garlan & Perry, 1995]. Domain specific architectures rarely pertain to a ‘pure’ architectural style, but instead involve a combination of several styles. These combinations can be manifest in various ways:

- a component system hierarchy may have an internal structure that belongs to a different styles;
- a connector hierarchy may be implemented internally according to a different style;
- a particular component may use various architectural connectors; and
- different styles may be used at different architectural levels of a system.

An example of a ‘mixed’ style architecture is C2 proposed in [Taylor et al., 1996] for GUI software. The basic properties of C2-compliant systems are substrate independence, pure asynchronous message-based communication, multi-threading, and lack of shared address space. Another instance of a ‘mixed’ style architecture is the OMG CORBA [OMG, 1995] which combines object-oriented, event-based, and client/server style elements.

2.2 Database Systems

A *database* is a collection of related data, i.e., known facts that can be recorded and have an implicit meaning. A database has the following properties [Elmasri & Navathe, 1989]:

- It is a logically coherent collection of data with some inherent meaning.
- It is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.
- It represents some aspects of the real world called the *mini-world*. Changes to the mini-world are reflected in the database.

Database management systems (DBMS) are general purpose software systems designed to manage large quantities of data. They support the definition of storage structures and provide data manipulation mechanisms. The main functional areas of a DBMS include the following [Abiteboul et al., 1995]:

- *Persistence*: data managed in DBMS should be persistent, i.e., its life span should extend beyond that of a particular database application so that it may be reused later.
- *Concurrency control*: the DBMS must support simultaneous access to shared information in an environment presenting a coherent database state to each database user.
- *Data protection*: the DBMS should provide integrity control mechanisms to prevent inconsistencies in the stored data, recovery and backup mechanisms to guard against hardware failures, and security mechanisms to prevent unauthorized user from accessing and/or changing sensitive information.
- *Secondary storage management*: DBMS manage amounts of data that are too large to fit in main memory. Thus, DBMS have to use various techniques to manage secondary storage such as indexing, clustering, and resource allocation.
- *Compilation and optimization*: the DBMS must provide translation mechanisms between the applications and the external and logical levels.
- *Interfaces*: the DBMS should provide interfaces to define the structure of stored data (data definition languages —DDL) and to manipulate the stored data (data manipulation languages —DML).

- *Distribution*: the DBMS must provide transparent access to multiple locally separated and heterogeneous information sources.

A *database system* (DBS) comprises the database and the database management software. A central assumption is the separation of the logical definition of data from the underlying physical implementation. Underlying the logical structure of a database is the concept of a logical *data model*. It is a collection of conceptual tools for describing the data, the data relationships, the data semantics, and consistency constraints. Several logical data models have been developed including the hierarchical, network, relational, and object-oriented. Due to the relevance of object-oriented and active database technology for our work we consider these in somewhat more detail in the following two sections.

2.2.1 Object-Oriented Database Systems

Object-oriented database systems (OODB) were developed in order to effectively support the database functionality needed by several classes of applications (e.g., engineering design) which requires more advanced data structuring concepts than those provided by relational database systems. Research in OODB started at the beginning of the 80's and led to the development of systems like Damokles [Dittrich et al., 1987] and O₂ [Babaoglou & Marzullo, 1993]. OODB support a data model which is based on a collection of objects. Objects are units of instantiation with a unique identity and a state. In addition to conventional database system, an OODB must provide a set of object-related features [Atkinson et al., 1992]:

- *Complex objects*: The OODB must support the construction of complex objects from simpler ones by applying the appropriate constructors. Basic operators such as retrieve, copy, and delete must be able to deal with complex objects.
- *Object identity*: The data model must support object identity so that objects exist independent of the value of their state. Two objects can be identical (they are the same object) or equal (the value of their state is equal).
- *Object encapsulation*: An object encapsulates both program and data. An object in an OODB thus has both a data part and operation part.
- *Types and classes*: Depending on the chosen approach an OODB should support either the notion of a type or that of a class. A *type* summarizes the common features of a set of objects. It consists of a type interface and a type implementation. A *class* is more of a run time notion and contains two aspects: an object factory used to create new objects and an object warehouse which refers to the class *extension*, i.e. the set of objects that are instances of the class.
- *Type or class hierarchies*: An OODB must support inheritance. Inheritance allows the factoring out of common specifications and implementations of objects. It can refer to structure and/or operations.
- *Overriding, overloading, and late binding*: An OODB must allow the redefinition of operation implementation for specialized types (*overriding*), the use of identical names for different implementations (*overloading*), and the choice of the appropriate implementation at run time (*late binding*).
- *Computational completeness*: The DML of an OODB must allow the expression of any computable function.

- *Extensibility*: The OODB must allow the definition of new types based on predefined types and must make no distinction between system-defined and user-defined types.

Additional extensions specific to OODB include type checking and inference facilities, extended transactions and version mechanisms.

2.2.2 Active Database Systems

Conventional relational and object-oriented database systems are passive as they only execute actions when explicitly commanded to do so through queries or update operations. A database system is called *active* (e.g., [Dittrich et al., 1995, Dittrich & Gatzia, 1996, Widom & Ceri, 1996]) when in addition to the “normal” database functionality it is capable of reacting autonomously to user-defined situations and then execute user-defined actions. The central concept of *active database systems* (ADBS) is that of *event-condition-action rules* (ECA-rules), also called *triggers*, through which the reactive behavior is specified. In general, an ECA-rule has the following operational semantics:

- when the defined event occurs,
- if the defined condition is valid, then
- execute the defined action.

The main extensions compared to the functionality of conventional database systems include the event and rule definition, as well as the rule execution. This functionality is briefly discussed below. For more extensive coverage of the topic we refer to the appropriate literature.

Rules in Active Database Systems

As already mentioned, an ADBS extends passive DBS by providing support for *reactive behavior*. This reactive behavior is specified by means of rules, i.e., the data definition language has operations to define rules. In their most general form, ADBS rules consist of three parts:

- *Event*: causes the rule to be triggered.
- *Condition*: is checked when the rule is triggered.
- *Action*: is performed when the rule is triggered and the condition evaluates to true.

The defined set of rules in an ADBS forms its *rulebase*. Once the rulebase is defined, the ADBS monitors the relevant events. For each rule, if its event occurs the ADBS evaluates the condition of the rule and if this is true it executes the rule’s action. Rule execution consequently may be performed for multiple rules at a time. ADBS features which are related to ECA-rules include operations to create, modify and delete rules, commands to deactivate rules (which are then not triggered by events until the symmetric activation command is executed), and mechanisms to define relative or absolute rule priorities.

Events in Active Database Systems

Events in ADBS denote the occurrence of situations of interest upon which a predefined reaction must be performed. An ADBS event can be conceived as a pair (*<event type>*, *<occurrence time>*) where *<event type>* denotes the description of the situation of interest and *<occurrence time>* represents the point in time when the situation actually occurs [Dittrich et al., 1995]. The designer of an ECA-rule event specifies only the event type. At run-time, multiple events of this type may occur.

Event types may refer to occurrences in the database system or its environment. Although no standard presently exists for the supported event types, most ADBS support the following types of events. Event types *internal* to the database system include:

- *Data modification* event types are specified based on the modification operations provided by the database system. In relational ADBS supporting SQL, these can be insert, delete, or update operations on a particular table. In case of object-oriented ADBS and depending on the provided DML, it may refer to constructor or destructor methods defined for a stored object type.
- *Data retrieval* event types are specified based on retrieval operations, for example, a select operation in active OODB supporting OQL.
- *Transactional* event types are specified based on transaction operations in the ADBS, such as begin or commit of a named transaction program.

Temporal event types which refer to temporal occurrences include:

- *Absolute temporal* event types specified as a specific point in time, e.g., 10:00 on March 24, 1998.
- *Periodic temporal* event types, as for example, “every Monday at 12:00” or “every hour”.

Event types which are defined in an application *external* to the ADBS include:

- *Application-defined* or *abstract* event types are specified by allowing the ADBS application to declare an event type which is then used in ECA-rules. The application notifies the ADBS of the occurrence of events of this type.

Event types in ADBS can either be primitive or composite. Primitive event types correspond to elementary occurrences and thus can be directly mapped to a point in time determined by the occurrence type. Composite event types are constructed by combination operators or *event constructors*:

- *Logical operators* such as conjunction, disjunction, and negation.
- *Sequence operator* denoting a particular occurrence order for two or more events.
- *Temporal operators* denoting event types defined in relation to another event type, as for example, an event occurring “10 minutes after E” or in case of distributed ADBS two events occurring concurrently.

Composite events are mapped to a point in time based on information about their component events. This mapping defines the formal semantics of event composition and is discussed in detail later in this thesis. *Event restrictions* may be defined for composite events to specify conditions that the component events must fulfill in order to form a legal composition. Restrictions may refer to *event parameters* —if supported by the ADBS— or other properties of the composite event (e.g., that all its components occurred within the same transaction).

Rule Conditions

Conditions in ECA-rules specify the conditions that have to be satisfied after the event has occurred in order for the action to be executed. Conditions are evaluated once the rule fires but their exact evaluation time depends on the rule execution model of the ADBS (see below). The following kinds of conditions can be distinguished:

- *Database predicates* are defined in the formalism for condition expressions supported by the database system, e.g., a where clause in a relational ADBS.
- *Database queries* defined in the database query language. The meaning is that the condition is true if the query produces a non-empty answer.
- *Calls to procedures or object methods* written in an application programming language which may or may not access the database. If the procedure returns a boolean value, then this is the value of the condition; otherwise, the meaning may be that if the procedure returns data then the condition is true.

If the rule language allows the definition of event parameters, then the condition can reference values bound to event parameters.

Rule Actions

Actions in ECA-rules are executed when the rule is triggered and the condition evaluates to true. The following kinds of actions may be supported by an ADBS:

- *data modification operations* written in the ADBS DML, e.g., object creation or non-constant method calls in an active OODB;
- *data retrieval operations* written in the ADBS DML, e.g., constant method calls in an active OODB;
- *database commands* such as transaction control operations (e.g., commit, rollback); and
- *application procedures and methods* which may or may not access the database.

Again, if the rule language allows the definition of event parameters, then the action can reference values bound to event parameters. Many ADBS allow action sequences to be defined in ECA-rule actions. We note finally, that actions may cause the occurrence of further events leading to cascaded rule triggering.

Rule Execution Model

The rule execution model prescribes how an ADBS behaves once a rulebase has been constructed. The behavior includes the semantics of rule processing and the interaction of rule processing with query and transaction processing. As a result, a large number of alternatives with respect to rule execution semantics exist. The dimensions along which a rule execution model is characterized include, for example, rule processing granularity, conflict resolution strategies when multiple rules are triggered, sequential vs. concurrent rule processing, and coupling modes. We consider here only this last dimension. For a detailed discussion of the other issues we refer to [Widom & Ceri, 1996].

In general, events occur within transaction boundaries and rules are executed within transactions. The transaction in which the event occurs is called the *triggering transaction*. If the rule executes in one or more transactions then these are called *triggered transactions*. The *coupling modes* [McCarthy & Dayal, 1989] determine the relation between rule processing and database transactions. They refer to the transactional relationship between the pairs (*<event triggering>*, *<condition evaluation>*) and/or (*<condition evaluation>*, *<action execution>*). Possible coupling modes include—but are not limited to—the following:

- *immediate*: further rule processing takes place immediately in the same transaction.

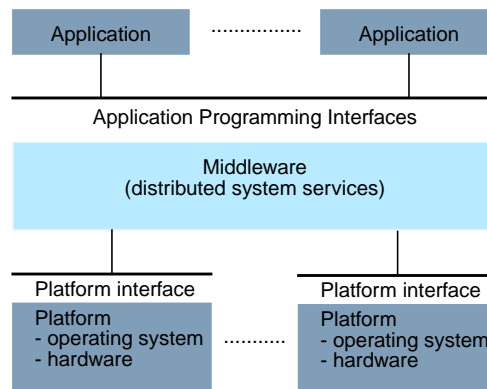


Figure 2-1: Conceptual positioning of middleware in a 3-tier architecture [Bernstein, 1996].

- *deferred*: further rule processing takes place at the commit point of the current transaction. This mode is useful for enforcing integrity constraints.
- *decoupled*: further processing takes place in a separate transaction. This mode can be subdivided depending on the commit-dependency of the two transactions:
 - *causally dependent*, meaning that the triggered transaction can commit only if the triggering transaction commits; and
 - *causally independent*, meaning that the triggered transaction is independent of the triggering transaction.

2.3 Information System Integration

The conceptualization and development of *middleware* is a result of recent trends in the construction and deployment of information systems. It is related to the universal use of computing technology in modern organizations, the connectivity achieved through digital communication technology, and the heterogeneity of connected systems. 'Middleware' is a term used in various contexts and with varying meanings. At the one extreme, middleware services denote an abstraction of distributed system services. At the other extreme, the term middleware however has been also used to describe all kinds of software which does not provide business-specific functionality.

2.3.1 Middleware Services

Middleware is often mentioned in connection with the concept of a three-tier architecture. According to this perspective, middleware is all kinds of software that lies between the operating system and networking software layer on the one side, and industry-specific applications on the other side (Figure 2-1). The characteristically pragmatic definition given by [Orfali et al., 1996] states that "*middleware [is] the slash (/) component of client/server*". In modern systems, middleware is replacing the non-distributed operating system functions with distributed functions that use the network [Bernstein, 1996]. The net result is that the programming interfaces provided by middleware define the computing environment of an application. [Orfali et al., 1996] distinguish between two classes of middleware (see Table 2-1):

- *General middleware* is a substrate present in most types of client/server interactions.
- *Service-specific middleware* is needed to accomplish a particular client/server interaction.

The characterization of a service as middleware has different qualitative aspects. It must potentially meet the requirements of various applications across different industries. Furthermore, a middleware service must have implementations that run on different platforms and be easily portable to other platforms. It has to be distributed in the sense that it either can be accessed remotely or that it enables other services and applications to be accessed remotely. Finally, a middleware service must be transparent with respect to a published API. Characterizing a particular set of services as middleware services is a moving target. A facility that is currently regarded as part of a specific platform may become middleware in order to be made available to further platforms. Conversely, middleware can be integrated into a platform to enhance the platform's value or for performance reasons.

2.3.2 Data, Control, and Process Integration in Workflow Systems

In this section we briefly refer to three concepts related to information systems integration: data integration, control integration, and process integration. We set our discussion in the context of application integration in workflow systems

Table 2-1: A non-exhaustive classification of middleware services, e.g., based on [Orfali et al., 1996].

<i>Middleware category</i>	<i>Service group</i>	<i>Service examples</i>
general purpose	network operating system	global directory
		network time
		distributed security
	communication	protocol stacks
		remote procedure calls
		messaging and queuing
special purpose	database access	SQL and SQL gateways
		data warehouse
	distributed transaction processing	transaction monitors
	coordination	groupware
	distributed object	CORBA services
		compound documents
	distributed system management	OSI-defined
		SNMP

Data Integration

Data integration refers to the sharing of (persistent) data among information system components. It ensures that all information in the system is managed as a consistent whole regardless of how its parts are manipulated by individual components. When considering the applicability of pure data-based application integration in workflow systems, we note the following shortcomings and limitations:

- no knowledge exists about which application in the workflow system performs a particular operation over the shared data;

- no explicit goal is defined for the manipulation of the shared data; and
- the management of the consistency between local data in applications and the global data in the workflow system is complex.

Control Integration

Control integration is concerned with component communication and interoperability to allow the flexible combination of their functionality as needed by workflows. It refers to the relative ease with which a new application can use existing services rather than duplicating them in its own code. Control integration has been pursued by researchers in collaborative engineering environments as an alternative to data integration due to the difficulties in agreeing on a common data model and the (perceived) limitations of database technology. Thus, control integration represents an effort to migrate functionality to where data resides.

A primitive level of control integration can be achieved using operating system scripts. For example, the pipe and redirection facilities in the Unix shell are often used to integrate small utility tools into a 'home-grown' application for some particular purpose. Higher levels of integration can be achieved when applications are able to communicate while they are running. A number of proprietary messaging systems have been used to enable custom-made tools to make and respond to requests for services and to send out notification of their actions. These include HP BMS (Broadcast Message System) in SoftBench [Cagan, 1990], and Sun ToolTalk. More recently distributed object messaging standards such as OMG CORBA architecture [OMG, 1995] have been developed with the intention of providing universal access to the interfaces of distributed objects.

Process Integration

Process integration is concerned with the role of processing entities in a workflow and ensures that workflow applications interact effectively to support this workflow. Process integration builds on data integration (to enable control data to pass through the workflow) and on control integration (to enable automated execution of workflow tasks). Process integration aspects are described in workflow specifications; process integration functionality is provided by the workflow execution infrastructure.

2.3.3 Database Systems as Middleware in Workflow Systems

Besides traditional database applications where databases are used for the storage and administration of production data, database systems have been used as the integration medium in coopera-

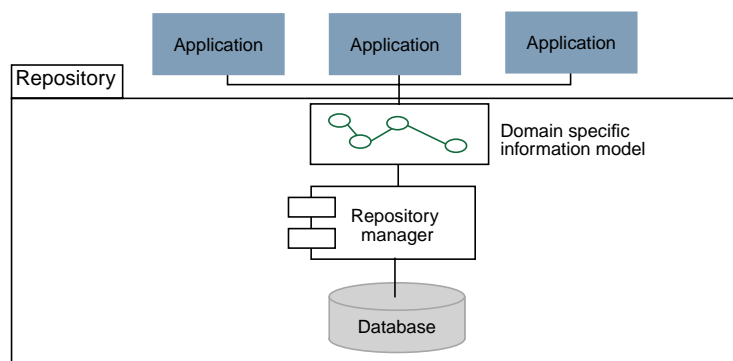


Figure 2-2: Conceptual repository architecture [Bernstein, 1998].

tive information systems. Data integration requires that at least a partial agreement exists among the various subsystems/applications in terms of a *domain specific information model*; in that case the term *repository* is used [Bernstein, 1998] (see Figure 2-2). The data in the repository can be owned by a specific subsystem or can be shared among various subsystems. Typical examples of repository-based data integration are software engineering environments. We note that often, due to limitations in available database technology, the developers of such environments had to develop customized repositories either from scratch or by extending an existing database system. A recent evaluation of database technology for process-centered software engineering environments can be found in [Barghouti et al., 1996]. We concentrate our elaborations in the rest of this section in defining the requirements on database functionality which concern workflow systems. The functional requirements on data integration facilities for workflow systems include extended information modeling, dynamic schema evolution, advanced concurrency control, version and configuration management, and distribution.

Data model. The data created and managed in a given workflow system has large variations with respect to the information granularity, the relationships between information granules, the intended access and modification patterns. The information which may be stored in a WFMS repository includes at least the following general categories:

- *definitions of workflows* in a form depending on the workflow specification language(s) that is/are used by the system;
- *application data* which is used for control flow; and
- *process instance data*, i.e., workflow execution and audit information.

Among these information elements different kinds of relationships may exist which also have to be adequately modeled. Examples of such relationships include inheritance, composition, versions, interchangeability, and usage. Type information about these information elements should also be easily accessible to all applications using the repository.

Further requirements for the data model depend on the formalism used for the internal representation of processes described in workflow schemata. The data model of a workflow system repository should allow the description of workflow specification elements at an abstraction level appropriate for the used workflow specification language. In addition, the database system should allow the efficient management of the workflow descriptions. When petri-net formalisms are used, for example, the database system should provide support for the efficient representation of places, tokens, transitions, and arcs.

Schema evolution. Facilities to change the definition of the structure of and operations on schema elements are required, i.e., the object type attributes and methods in an object-oriented repository. These facilities are subsumed under the term *schema evolution*. Changes in the definition of schema elements have potentially widespread consequences affecting both the actual data as well as related —still unmodified— schema elements. Schema evolution is particularly important in workflow systems and is required to accommodate new requirements on the representation of workflow information. For example, additional workflow auditing data may be required as a result of new legal constraints. This requires that the database schema describing executing workflows must be extended accordingly.

Versions and configuration management. Due to the evolution of the descriptions of workflows and of workflow application data, configuration management functionality has to be provided. Often multiple versions of the application data may have to be stored and accessed in parallel by

different workflow schema versions. Also, different versions of a particular workflow can be enacted at any time. Configurations are built consisting of various artifacts of different versions and have to be efficiently managed. Some systems capture configuration semantics in relationship types (e.g., composite links in PCTE [Wakeman & Jowett, 1993]). The underlying DBMS has to support the derivation of new versions, the removal of subversions, the automatic merging of versions, and the maintenance of version histories.

Distribution and heterogeneity. Typically, workflows execute over multiple interconnected workstations, among different local networks, even on different operating systems. Thus a requirement on DBMS is their ability to operate in distributed heterogeneous environments.

Efficient storage. A final requirement we consider stems from the nature of the data stored and manipulated in workflow systems. The underlying DBMS should allow the efficient storage and retrieval of different kinds of objects of highly varying sizes and types.

2.3.4 Distributed Object Middleware

In recent years, two competing middleware architectures have emerged: OMG's *Object Management Architecture (OMA)* [OMG, 1995] and Microsoft's *Distributed Component Object Model (DCOM)* [Orfali et al., 1996]. We briefly consider OMA and note that despite the different mechanisms provided by DCOM, the underlying paradigm is similar.

Object Management Architecture

The Object Management Group (OMG) is a consortium established in 1989 has as its goal the development of a common architectural framework, the OMA, for object oriented applications based on widely available interface specifications. OMA includes a reference model which describes the components of a distributed object system:

- The *Object Request Broker (ORB)* [OMG, 1995] is an object message dispatcher which enables distributed objects to send and receive requests and responses.
- *Object Services* is a collection of objects and their interfaces which provide basic object implementation functionality. The services defined by OMA include naming, concurrency control, externalization, time, persistence, event management, licensing, life cycle, query, relationships, transaction, collection, security, and trader. The standardization of these interfaces was completed by the end of 1997 [OMG, 1997].
- *Common Facilities* is a collection of end-user-oriented services useful across different application domains, such as scripting, compound documents, and workflow.
- *Domain Objects* are application domain-oriented services for specific industries.
- *Application Objects* are finally specific to a particular application.

The *Common Object Request Broker Architecture (CORBA)* [OMG, 1995] defines the architecture of distributed object-messaging middleware. It provides the substrate for location transparent message exchange between objects whose interfaces are described in the *CORBA Interface Definition Language*¹. In addition to the standard documents, the CORBA architecture has been described extensively in many books and articles. In the following subsections we briefly mention

¹ OMG's IDL is currently in the process of becoming an ISO standard.

the main aspects of OMA related to our own work: CORBA, event, notification, and ECA-rule service.

CORBA. CORBA is a domain-independent distributed programming environment. The CORBA application developer has to write code in which low-level details have to be considered which essentially lie at the abstraction level of class libraries. The abstraction layer of IDL roughly corresponds to that provided by C++ header files. In effect, its mapping to various object-oriented programming languages is pretty straightforward. While this is important for the implementation and adoption of the standard, it limits the potential of IDL as a specification language.

Event and Notification Service. OMA defines basic asynchronous communication functionality in the Event Service specification [OMG, 1997] described in the next section. Due to its very low level of abstraction, an extension of the basic event service, the Notification Service, is under consideration at the time of this writing [OMG, 1998a].

ECA-rule Service. ECA-rule services have not yet been defined by OMG at the time of this writing. However, the need for higher level rule management services in OMA has been identified in the research where its use for building distributed active database applications is discussed [von Bültingsloewen et al., 1996]. In this approach, autonomous information sources such as relational or object-oriented database systems are integrated in the environment by means of wrappers which detect events in these information sources. These events are forwarded to a rule system which performs composite event detection without however, supporting global temporal event ordering. In our work, we consider how ECA-rules can be used for distributed workflow execution providing global temporal event ordering.

2.4 Event-Based Integration

The term “event” is used in many domains of computer science to capture the notion of an *autonomous asynchronous occurrence*. This is in contrast to the term “message” which usually captures the notion of an exchange of information between a specific message sender and a specific message receiver. The meaning of the terms however, is clear only in the context of the supported interaction paradigm: events are usually associated with *implicit invocation* while messages are associated with *explicit invocation*.

The conventional component interaction paradigm used in large systems is referred to as explicit invocation and refers to interaction through mechanisms such as (remote) procedure call (e.g., as in OSF DCE) or distributed message-passing (e.g., in CORBA [OMG, 1995]). The main characteristics of this integration paradigm which supports a tight coupling between participating entities, can be summarized as follows [Griffel, 1998]:

- *exactly one* receiver exists for every component interconnection instance;
- the sender has to choose this receiver; and
- if the receiver is not reachable *during the call*, the interaction fails.

A different technique proposed for loose system integration is implicit invocation, also called reactive integration or selective broadcast over an *event channel* which is a container of asynchronous messages or events. Implicit invocation systems were recognized as defining a specific architectural style and were formally described in [Garlan & Notkin, 1991].

In an implicit invocation system the actions performed by one component may cause the invocation of operations in other components without the original component having *explicit static references* to those components. Thus an implicit invocation mechanism is a collection of components each of which has an interface that specifies a set of methods and a set of events. The methods define operations that other components can explicitly invoke; events define actions that the component promises to announce to other components in the system. Implicit invocation includes the concept of *registration*. Other components in the system register an interest in an event by associating a procedure with the event. When the event is announced, the system itself invokes all of the procedures that have been registered for the event. Consequently, an event announcement “implicitly” causes the execution of procedures in other components. This indirection allows the decoupling of events from components unlike, for example, events in Java Beans [Sun Microsystems, 1997] which are bound to an event listener implementation. Thus the connectors in an implicit invocation system may include both procedure call and associations between procedure calls and event announcement.

Event-based software integration has a long and successful utilization record in computer science. The spectrum of its use includes the following types of applications:

- In *GUI event-based programming*, applications wait in an event loop for user generated events (e.g. keyboard input or mouse clicks). An example of this approach is the Apple Macintosh operating system [Apple Computer, 1991].
- In event-based *distributed debugging* systems the debugging processes often appear as performance impairment rather as erroneous state. In such cases, it is important to analyze the behavior of participating software systems. For a detailed discussion of events in distributed debugging systems see [Schwidorski, 1996].
- In *distributed process control systems* events have been used to denote the changes to the environmental conditions. [Kirmann et al., 1986] for example, describe a process control architecture built around a multi-master event bus which supports event broadcasting. Events correspond to changes of values in boolean functions of environmental variables or the achievement of a certain process state.
- *Event services* are also provided within primarily message-based integration architectures, such as OMG's CORBA, Microsoft's DCOM, and Sun's JavaBeans (see below).
- *Tool integration* through events has been pursued in software development environments, such as FIELD or HP Softbench (see below).
- Generic event-based software *integration frameworks* such as Yeast and Polyolith (see below) provide a generic platform for the integration of software components.

In [Barret et al., 1996] a framework for the systematic characterization of aspects of event-based software integration is described. Systems are characterized along five dimensions:

- Two primary *methods of communication* are distinguished: in *point-to-point communication*, data is sent directly from one software module to another as for example in procedure calls or application-to-application communication; in *"multicast" communication*, software modules express their interest in receiving certain types of data that are routed by a server (*selective broadcast*) or software modules have their inputs and outputs bound to the channels of an abstract bus (*software bus*).
- The degree of *expressiveness of module interaction descriptions* which can lie in a continuum between no explicit specification, through procedure signatures, to specification of

the module actions on receipt of particular types of messages and even of connections between modules.

- The *intrusiveness of module interaction descriptions* which may range from the modification of module source code, to wrapping or encapsulation.
- Support for *static vs. dynamic specification* of module behavior and interaction.
- *Naming issues* which define the degree of awareness of the sender of the names and locations of the recipients.

[Barret et al., 1996] define in their framework a type model in which functional components of event-based systems are characterized based on various type-specific attributes. The component types considered include events and messages, informers (event and message generators) and listeners (event and message recipients), registrars, routers, message transforming functions, delivery constraints, as well as composition by groups of functional components. In their paper, [Barret et al., 1996] use this framework to describe the three systems FIELD, Polyolith, and CORBA.

2.4.1 Event-Based Integration in CORBA, DCOM, and Java-Beans

Simple event-based component interaction is supported in two competing distributed system architecture models, OMG's CORBA and Microsoft's DCOM. The purpose of event services in these architectures is asynchronous object communication. The *CORBA Event Service* [OMG, 1997] distinguishes between an event supplier and an event consumer (which would be the situation detector in the approach of [von Bülzingsloewen et al., 1996]). The supplier and consumer communicate through a single well-known *event channel*. Two approaches towards initiation of event-based communication are distinguished in CORBA:

- In the *push model* a supplier of events initiates the transfer of event data towards the event consumers. Figure 2-3 depicts the conceptual architecture of the CORBA event service push model¹.
- In the *pull model* the consumer requests the event data from a supplier. The communication can be *generic* in which case all event data is packaged to a single parameter, or *typed* in which case event data is passed by means of defined parameters. Typed event channels can provide filtering based on the event type. Application data are passed as an event parameter. A single channel can handle any combination of approaches: push/pull and generic/typed. Complex events involving multiple objects are handled by constructing a notification tree of event consumers/suppliers checking for successively more specific event predicates. The CORBA event service specification does not define quality of service properties with respect to the reliability of event delivery (from "at-most-once" to "exactly-once"), scalability, availability, throughput, and performance.

An extension of the event service is under consideration at the time of this writing. The new Notification Service [OMG, 1998a] adds various capabilities to the Event Service such as structured event types, client registration to specific events, performance enhancements, and an optional event type repository which can be accessed by event consumers and suppliers

¹ In the subsequent discussion we use simple diagrams to visualize essential elements of system architecture. These diagrams are kept as simple as possible and consist of two types of objects: components (boxes) and connectors (arrows).

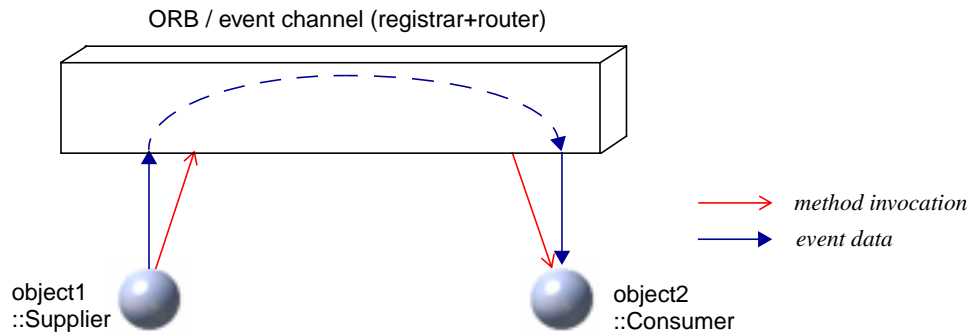


Figure 2-3: The conceptual architecture of CORBA event services push model. The producer (object 1) pushes an event to the event channel which pushes it to the consumer (object 2). The pull model is analogous but method invocation is in the opposite direction with respect to event data.

DCOM [Orfali et al., 1996] allows the specification of outgoing parameterized events in event generating *source components* and provides interfaces for the subscription to incoming events by event-handler *sink components*. This allows a rudimentary event notification mechanism in which producers and consumers of events are directly connected via sinks and sources. There is thus no concept analogous to that of an event channel. Furthermore, only primitive events are supported, such as, object data modification, and object renaming.

Finally, in JavaBeans [Sun Microsystems, 1997] events are implemented as plain vanilla Java objects. They are exchanged between one source and multiple consumers. In JavaBeans there is no concept analogous to that of an event channel. The decoupling between sender and receiver is limited as the event source component must have references to the Listener-interface of the event consumers. Furthermore, event notification across multiple Java Virtual Machines is not directly possible [Griffel, 1998].

2.4.2 Event-Based Integration of Tools

In section 2.4.1 we considered the event-based integration mechanisms provided by the principal component technologies. In this section we consider how events are used for tool integration in *software engineering environments*. The emphasis in such systems lies on “white-box” control integration [Valetto & Kaiser, 1996]. In that case, a custom tool is explicitly developed as part of a particular environment or alternatively, the source code of a legacy tool or application is modified to match the environment’s interface. White-box integration has been used in various research and commercial software engineering environments. Tools developed according to domain-specific standards such as the Portable Common Tool Environment [Wakeman & Jowett, 1993] or more general interoperability standards such as CORBA [OMG, 1995] efficiently support white-box integration.

FIELD

FIELD [Reiss, 1995] is a collection of communicating tools for code management, editing, compilation, debugging of programs. The integration goals of FIELD are the following:

- Tools can interact directly. If for example, a user wants to set a break-point in the editor, the editor must be able to issue a corresponding debugging command.

- Tools share dynamic information, i.e., the different environments need to know the current execution context. The tools might also have to know the execution state of other tools, e.g., the *make* utility may start a recompilation automatically when the editor has saved a file —providing rudimentary process support.
- Consistent access to the program’s source code is provided, independently of the access purpose (e.g., editing, or setting break-points).
- Static, specialized information is available to all tools. It includes rules needed to build a particular program, cross-reference information, profiling data, and information about the program execution environment. This information must be available to the various system components on demand and with an up-to-date content.
- The environment should accommodate existing tools in an efficient way and without requiring (extensive) tool modification.

Communication in FIELD is based on the *selective broadcasting* of synchronous command requests —equivalent to inter-tool command invocations— and asynchronous events containing data known to one tool which might be of potential interest to other environment components —notification. Both kinds of messages are forwarded by the sender to a centralized message server which broadcasts them to registered recipient tools. The message facility uses a client library linked into each FIELD tool and a message server process. Each tool and service is started by initializing the client library which opens a connection to the message server. Subsequently through this library the tool registers patterns describing the messages it is interested in. As the tool runs it sends ASCII strings to the server. The message server of FIELD handles these strings and string-based pattern matching is used to determine which clients receive the broadcast.

In addition to synchronous and asynchronous messages, the message server provides two special classes of messages: *priority messages* allow a higher-level message interface (the policy tool) to modify, delete or insert messages into the message stream. Such messages are handled as synchronous messages by the server before any other incoming messages and their replies can be handled by the server in special ways. *Default messages* provide fall-back handlers invoked when synchronous messages are matched that either had no responders or to which all replies were NULL. They are used to invoke tools in FIELD automatically.

The conceptual architecture of FIELD is depicted in Figure 2-4. The message server provides communication and integration of the various component tools. Native FIELD message-based

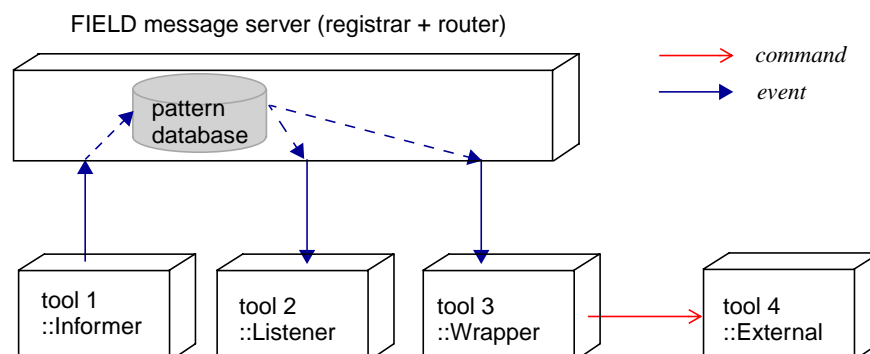


Figure 2-4: The conceptual architecture of event-based integration in FIELD. Tool 1 sends an event message to the server which “multicasts” it to tools 2 and 3. Tool 3 calls a service of the wrapped external tool 4 as a result.

tools may be used to wrap non-message based tools standard UNIX tools. Selective broadcasting is achieved by making each message-based tool notify the message server about the events it is interested in receiving, specifying the command requests it can understand and information messages it will want to act on.

FIELD was the first environment to propose a tool integration paradigm based on implicit invocation. Its limitations concern mainly the level of integration it provides which is essentially limited to a lexical level with the exchange of simple strings which have no semantic content whatsoever but instead have to be understood by the communicating peers. Additionally, FIELD is a centralized environment targeted toward programming and as such difficult to scale up to multi-personal project support. Various systems use direct extensions of the integration paradigm of FIELD the most prominent being HP Softbench [Cagan, 1990] which introduced the notion of tool protocols which are standard sets of operations and information messages and DEC FUSE [Hart & Lupton, 1995] in which messages are assembled by using an interface that resembles a remote procedure call.

Polylith

Polylith [Purtilo, 1994] is a software integration framework intended to allow the interconnection of heterogeneous software components written in different programming languages and executing in a distributed environment. The principal goals of Polyolith are the following:

- independency of the implementation and the interface of components;
- independency of component execution location from their implementation, i.e. location transparency; and
- abstract specification of component communication mechanisms.

Polyolith combines the event-driven approach with “white-box” tool integration through *fragmentation*. Tools are identified by services called *modules* — and connect their input and output ports to an abstract communication agent —the *software bus*— in order to send and receive messages on named bus channels. Entire external tools can also be integrated by relinking with libraries which provide access to the system kernel interface. Polyolith supports both point-to-point and multicast-based communication (see Figure 2-5).

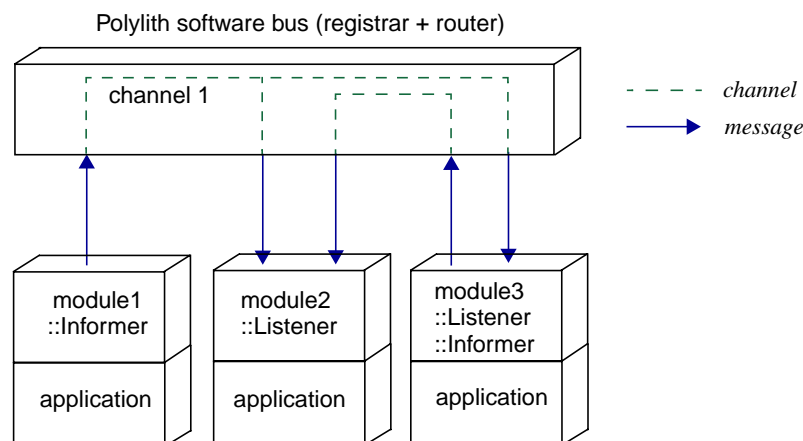


Figure 2-5: The conceptual architecture of Polyolith message-based integration. A module 1 sends a message on channel 1 which “multicasts” it to modules 2 and 3. Module 3 sends a message to module 2 on a different channel.

Yeast

Yeast [Krishnamurthy & Rosenblum, 1995] is a client-server system developed to provide general purpose event-action services. Clients register event-action specifications with the centralized server. The server performs event-detection and triggers the corresponding action of a specification (see Figure 2-6). Yeast provides a global event space shared by all users. The generality of Yeast has been an explicit goal of the system developers, a fact reflected in the requirements they defined which include:

- the system must have knowledge about external events;
- there should be no restriction on the actions that can be performed;
- both temporal and non-temporal events must be handled;
- new kinds of events must be definable; and
- it should provide a persistent state.

The event language of Yeast allows the specification of temporal events and events which refer to the modification of attributes of predefined or user-defined objects. Typical objects in Yeast are files, directories, terminal devices, and processes. Composite events can be defined using sequence, conjunction, and disjunction operators. Actions in event-action specifications are commands executed by a command interpreter.

The main limitations of Yeast concern its centralized server architecture, the use of environment polling for event detection, and the low level of abstraction of the provided middleware services. The generality of Yeast also belies the fact that there is no identified semantic content in the event or action specifications and thus it cannot be considered as a provider of middleware services but rather like a more powerful demand-driven system such as the *make* build utility. In this sense, Yeast defines event providers but no event recipients.

2.4.3 Events in Workflow Systems

In this section we consider the use of events in workflow systems. We concentrate on the conceptualization of events and refer the reader to section 3.3.3 for a discussion of ECA-rule-based workflow specification. In general, we can distinguish between two uses of events in existing workflow system architectures:

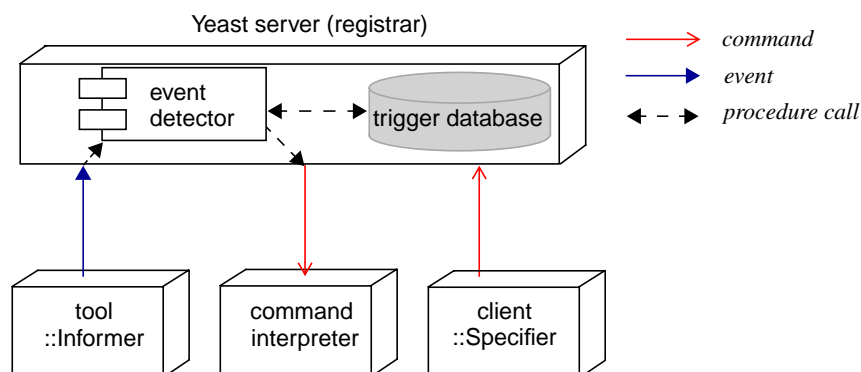


Figure 2-6: The conceptual architecture of Yeast event-based integration. A tool action generates an event detected by the server which executes the actions of matching triggers. The Yeast client defines trigger specifications managed in Yeast.

- Events are generated by the workflow engine and are used to trigger workflow engine rules whose actions drive further workflow execution. They represent changes in the workflow execution state. workflow systems which use an ADBS as the underlying workflow enactment engine follow this approach.
- Events are used as the interaction mechanism between environment subsystems. They are used to exchange control and data information. Event-based integration as described in sections 2.4.2 and 2.4.3 belong to this category. ADBS functionality can also be used in this case, e.g., for event management and detection, but it provides the supporting technology and is not the workflow enactment engine in itself.

In both kinds of systems temporal events are often used to provide a notion of real-world time into workflow execution. The efficient support of workflow management applications by ADBS requires the definition of abstract events (see section 2.2.2). When this is not the case, interesting occurrences must be expressed by other kinds of events, such as changes of database tables (e.g., in the Panta Rhei system [Eder & Groiss, 1996]). This represents an unnecessary indirection and may additionally lead to performance problems concerning event detection. In order to control workflow activities, composite events must be supported to express more complex dependencies, such as, and-joins. Furthermore, temporal events must be provided to express time-related situations and time-relationships between activities.

Workflow systems which use an ADBS-based enactment engine are considered first. Typically, the ADBS is used for the management of workflow execution data. For example, workflow instances are stored in the database as objects in TriGS_{flow} [Kappel et al., 1998] or entries in relational tables as in CapBasED-AMS [Karlalalem et al., 1995]. In TriGS_{flow}, executing workflows are represented by objects in an OODB. Two kinds of primitive events are distinguished: message events which are associated to the sending of a message to an activity object and time events such as those proposed in the SAMOS ADBS [Gatzju, 1995]. Message events either refer to the point before the requested method is executed (keyword *pre*) or after the method has finished execution (keyword *post*). In that case, database events refer to the changes to the state of executing workflow instances, i.e. the progress of the workflow. Similarly in CapBasED-AMS [Karlalalem et al., 1995], events occur when operations in the underlying relational ADBS are executed. These operations manipulate activities and agent properties stored in relational tables. In addition, temporal events and external events explicitly triggered by the system users are provided. The external events have to be registered with the workflow engine. The use of ADBS for coordination in cooperative information systems is also proposed in [Berndtsson et al., 1997]. In essence, primitive events correspond to events in the state diagrams which are used to describe agent behavior. Thus events in the state diagrams are mapped to method events of objects representing them in the underlying ADBS.

[Bussler & Jablonski, 1994] represents one of the first efforts to use ECA-rules for agent coordination in workflow systems. The proposed approach uses externally generated events for agent notification and synchronization; control operations on tasks generate events. The provided event types are named according to the operation they refer to: start, abort, reset, and finish.

A hybrid approach is used in [Casati et al., 1996]. *Workflow internal events* correspond to state-transitions during task execution within the workflow engine: start_{case}, end, cancel, suspend, resume, execute, and refuse are distinguished. *Workflow external events* reflect changes to the shared database (insert, delete, and update) or temporal events. In more recent work on the WIDE project [Ceri et al., 1997], events are primarily used to describe exceptional situations and are used in exception handling rules. Beside temporal events referring to instants, intervals and periodic occurrences, workflow internal events are classified in *regular exceptions* (e.g., constraint vi-

olations or workflow variable updates), application-defined *alarms* (e.g., task execution delays), and *execution exceptions* (e.g., task cancellation, agent unavailability). Workflow external events signal the arrival of documents, e-mails, and telephone calls(!). It is doubtful, however, whether the richness of event types in WIDE provides efficient assistance to the developer of workflow applications.

In the OPERA distributed coordination system [Alonso et al., 1997], events are used to externalize the intermediate results of activities. They are effectively abstract events which have to be declared in the system. Exception events are raised by tasks when unexpected situations occur or when external intervention is required for the continuation of processing.

Finally, we mention at this point the interesting work by [Jasper & Zukunft, 1997] which propose the use of so-called active abstract data types (AADT) to extend the interfaces of passive database objects with parameterized atomic event types and atomic actions, essentially allowing asynchronous object communication within the ADBS. When using AADT for workflow management, workflow activities are represented by AADT and the supported event types refer to the starting and ending of activities. It is not clear however, how distribution is supported by the underlying ADBS AIDE.

2.5 Summary

Workflow management is a multi-disciplinary domain in information systems. WFMS have been characterized as middleware by some authors. For the implementation of workflow systems, however, various lower-level middleware services can be used such as distribution, persistence, and notification. We consequently considered some basic technologies which are particularly relevant to our own work:

- Database management systems in general and ADBS in particular, provide a basic platform for support workflow system development and operation.
- Distributed object technology can be used to facilitate —to a certain extent— distribution of workflow systems.
- Event-based system integration represents the underlying paradigm in our work for the description of workflow system architectures. In various existing systems, events are used to support asynchronous interaction between components. Furthermore, ADBS have been used for the implementation of workflow execution engines.

3 The Workflow System Domain

In this chapter we define the domain of workflow systems. We introduce the term of business process reengineering and consider how the operational requirements of modern businesses have led to the widespread use of workflow management technology. In section 3.2 we discuss the phases followed from the redesign of business practices to workflow system development. In section 3.3 we discuss the conceptual workflow metamodels and workflow specification languages. Finally, in section 3.4 we describe two efforts to develop a reference architecture for workflow management systems and survey metamodel concepts for the description of workflow system components and applications.

3.1 The Business Process Perspective

The globalization and rapid change in today's business environment are facts with which every business organization has to cope in order to ensure its long-term survival. Companies have to continuously adapt to a changing business environment. This environment consists of various elements with which informational, financial, human resource, machine, and material flows take place [McLeod, 1994]. The elements of the business environment include customers, suppliers, stockholders or owners, labor unions and the labor market, government agencies, financial institutions, as well as competitors. In order to cope with their dynamic environment, firms strive to improve their efficiency in processing and managing the mentioned flows. They particularly tend to streamline their organizational structures building more efficient flat organizations [Drucker, 1988] and improve the efficiency of their *business processes* by careful reengineering [Hammer & Champy, 1994].

Business processes define how things have to be done in an organization by defining the *steps* which have to be performed. They are generally identified in terms of beginning and end points, interfaces, and elements of the business environment involved, particularly the customers. Examples of processes include developing a new product, ordering goods from a supplier, creating a marketing plan, and processing and paying an insurance claim. Business processes may be defined based on three dimensions [Davenport & Short, 1990]:

- *Entities*: Processes take place between organizational entities. They could be inter-organizational, inter-functional, or inter-personal.
- *Objects*: Processes result in manipulation of objects. These objects could be physical or informational.
- *Activities*: Processes could involve two types of activities: Managerial (e.g., develop a budget) and operational (e.g., fill a customer order).

In this context the concept of *quality* [Geiger, 1994] is often used to describe how business processes should be designed. Quality denotes the relation between required properties and achieved properties of the business process; these properties can be either quantitative (e.g., maximal duration) or functional (e.g., completeness). *Business process reengineering/redesign (BPR)* has as its

objective the optimization and quality improvement of business processes. Business process redesign has been defined by [Davenport, 1993] as

“the analysis and design of workflows and processes within and between organizations”

BPR involves the critical analysis and redesign of existing business processes to achieve improvements in performance measures. This does not necessarily imply the increased use of IT but rather its rationalized, efficient support of business processes [Österle, 1996]. Special emphasis is given to the so-called “critical business processes” which are important for the survival of the firm and define its core competence. They usually refer to the provision of products and services to customers.

Although IT is not the driver for BPR, process redesign can create new opportunities and application domains for information technology. IT is rather considered an enabler for business process redesign and process innovation [Davenport, 1993]. More specifically, Davenport identifies the following ways in which IT may enable and affect process innovation:

- *automational*: elimination of human labor and achievement of more efficient structuring of processes;
- *informational*: capturing of process information for purposes of analyzing and better understanding;
- *sequential*: transformation of sequential processes to parallel in order to achieve cycle-time reductions;
- *tracking*: monitoring the status of executing processes;
- *analytical*: analysis of information and decision making;
- *geographical*: allowing the organizations to effectively overcome geographical boundaries;
- *integrative*: improvement of process performance by moving from highly segmented tasks to a “case management” approach supported by information technology;
- *intellectual*: capturing and distribution of employee expertise; and
- *disintermediating*: increasing efficiency by eliminating human intermediaries in relatively structured tasks.

Different kinds of IT impact process innovation in different ways. Computer aided design and engineering systems, for example, support product development processes, expert systems support analysis and decision processes, asset management systems are used to optimize the use of key assets such as physical goods or financial assets in business processes. As we will discuss in the next sections, the single most important process enabling information technology is workflow technology and workflow systems.

3.2 From Business Processes to Workflow Systems

The actual improvement in competitiveness and productivity through BPR depends to a large degree on the successful implementation of process automation. In this section we briefly describe this transformation process and the difficulties encountered. For a more extended discussion we refer the interested reader to specialized literature on the subject (e.g., contributions in [Österle & Vogler, 1996], [Gaitanides, 1994], and especially [Schäl, 1996]).

There are various approaches for the implementation of business process change through workflow technology. In [Davenport & Short, 1990] and [Davenport, 1993] a phased sequential approach for the implementation of process change is proposed which is technology-neutral although IT can play a supporting and enabling role in all the phases:

- (1) Develop the business vision and process objectives: BPR is driven by a business vision which implies specific business objectives such as cost reduction, time reduction, quality improvement, etc.
- (2) Identify the processes to be redesigned: Most firms focus on the most important processes or those that conflict most with the business vision. Lesser number of firms use the exhaustive approach that attempts to identify all the processes within an organization and then prioritize them in order of redesign urgency.
- (3) Understand and measure the existing processes thus avoiding old mistakes and providing a baseline for future improvements.
- (4) Identify IT levers: Awareness of IT capabilities can and should influence process design.
- (5) Design and build a system to support the new process: The actual design should not be viewed as the end of the BPR process. Rather, it should be viewed as a prototype, with successive iterations. The metaphor of prototype aligns the BPR approach with quick delivery of results, and the involvement and satisfaction of customers. Successive iterations should lead to a mature workflow system.

Phases 1 to 4 serve the analysis and definition of the business aspects and the organizational context of the workflow system. Phase 5 involves the methodical use of IT to define the new process and transform it to a formal workflow model. It includes the construction of a workflow system which involves the integration of the existing heterogeneous information systems and legacy applications. As such it can be considered to encompass all activities in the information system development life-cycle or software process model (as described in e.g., [McDermid & Rook, 1991]).

A plethora of methods and tools have been proposed to support developers of workflow systems in the implementation of business processes with workflow technology, that is, the design of workflow applications through the modeling of workflows. This corresponds to the development of application-level views of a workflow system as defined in chapter 1. A simple conceptual metamodel of these methods is depicted in Figure 3-1. A phase covers a period of time in which activities are performed which contribute towards the achievement of a goal, i.e., the implementation of the new business process by a workflow system. Note that, although precedence relationships exist between individual phases no strict sequencing is assumed between them. Development methods consist of phases in which documents are prepared, which in turn are used in other phases. Documents are relevant to the people perform the various activities of individual

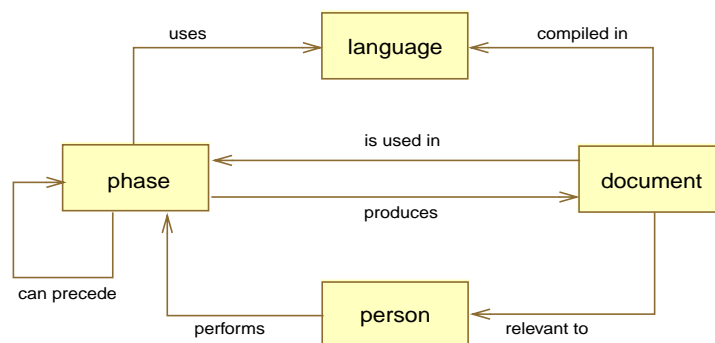


Figure 3-1: Conceptual metamodel for workflow system development [Jablonski et al., 1997].

phases. Thus, technical documents are relevant to development engineers, while abstract, conceptual documents interest management. Technical documents are compiled in more or less formal languages. These languages are the subject of our consideration in the next section within the framework of workflow specification.

3.3 Workflow Specification

As mentioned in the introduction, a basic assumption of our approach is that of the existence of various abstraction levels at which a workflow system can be described. In this section we consider the development of application-level views by means of workflow specification languages. The domain-oriented abstractions underlying a workflow specification is called a *workflow meta-model*¹. Considering the repository architecture depicted in the previous chapter, the workflow metamodel is the conceptual schema for the repository that holds workflow relevant data. It is conceptual rather than logical because the metamodel does not take the implementation technology into account, and thus needs to be mapped into a logical or physical schema just like a regular conceptual schema. The metamodel serves as the basis for defining workflow modeling languages. In other words, the representational concepts of a workflow metamodel are expressed in a *workflow modeling* or *specification language*. By means of the constructs of the workflow modeling language, *workflow specifications* or *models* can be constructed. In this section, we briefly discuss workflow metamodels as they provide an appropriate conceptualization of the workflow management domain.

3.3.1 Workflow Metamodels

[Georgakopoulos et al., 1995] distinguish between activity-based and communication-based workflow metamodels. In *activity-based metamodels* the focus lies on the work to be performed. The following conceptual elements are usually provided:

- *tasks* are partial or total orders of operations, descriptions of human actions, or other tasks; *workflows* are partial or total ordering of sets of tasks;
- *manipulated objects* can be documents, data records, images, printers, etc.;
- *roles* are placeholders for human capabilities or information system services required to perform a particular task;
- *agents* or *actors* are humans or information systems that fill these roles, perform tasks, and interact during workflow execution.

In *communication-based metamodels* the communication among the various actors is the central perspective of the metamodel. One of the few systems which are based on a communication-based metamodel is ActionWorkflow [Medina-Mora et al., 1992]. The principal conceptual element is the *coordination action* which is a closed loop which proceeds in four phases: proposal, agreement, performance, and satisfaction. Additionally a *customer* and a *performer* are the communicating *actors* in a coordination action. Communication-based metamodels are used mainly in computer supported cooperative work (CSCW) systems.

¹ The terminology frequently used in workflow management literature has been inspired by database research and does not correspond to that used in other software engineering literature which is closer to the mathematical notion of a model. Our use of the terms “model” and “metamodel” corresponds to the broadly accepted use in software engineering.

[Curtis et al., 1992] and [Jablonski & Bussler, 1996] have systematically classified the various aspects of workflow modeling. The approach of [Jablonski & Bussler, 1996] in particular, emphasizes the separation of concerns during workflow management. They propose the following (quasi orthogonal) perspectives of a workflow:

- The *structural* perspective describes the workflow activities, i.e., what has to be executed and their internal structure, if any.
- The *behavioral* perspective describes the execution sequence and dependencies between structural elements, i.e., the control flow. It determines how the routing of work proceeds. Conceivable routing types include the following:
 - in *sequential* task execution one task is followed by the next task;
 - in *parallel* task execution two tasks are executed at the same time in any relative order;
 - in *conditional* task execution either one or the other task is executed, but not both (exclusive-or);
 - in *iterative* task execution a task is executed multiple times.
- The *informational* perspective describes the information structures and flows within the workflow system. It refers to information which is used for control flow by the workflow management system
- The *organizational* perspective describes to which organizational units the different workflow actors belong, the subordination, cooperation, and substitution relationships between the organizational units and between the actors, and the relationships to people which do not directly participate in the workflow. It also describes the assignment of workflow activities to actors.
- The *operational or workflow application* perspective describes how applications execute a particular activity and the resources used during that execution. The operational aspect concerns the description of workflow applications.

The WfMC Workflow Metamodel

Activity-based modeling is central to the Workflow Management Coalition (WfMC) Reference Model [WfMC, 1994, WfMC, 1998]. The Workflow Process Definition Metamodel [WfMC,

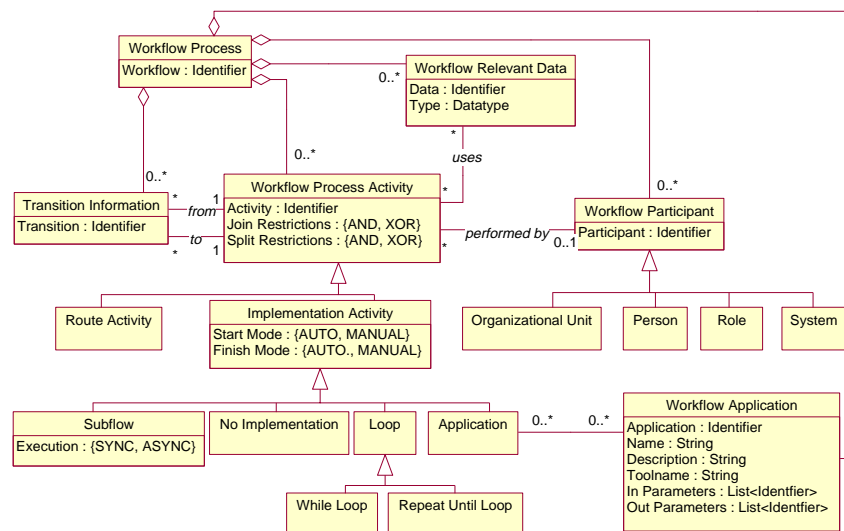


Figure 3-2: Class diagram [Booch et al., 1999] of the WfMC Workflow Process Definition Metamodel.

1998] has been developed mainly with the objective of allowing the interchange of process definitions among different workflow management systems. It defines the following elements (see also Figure 3-2): a *workflow process* consists of several steps called *activities* which use *workflow relevant data* and may have related *transition information* governing the transitions from one activity to another. The transitions may refer to the workflow relevant data passed to and produced by activities. Data elements have a name and a type associated with them. The activities may involve the invocation of *workflow applications* which themselves may use the workflow relevant data. We consider the WfMC workflow specification metamodel in more detail in chapter 7.

Variations of the WfMC workflow metamodel underlie, for example, systems such as the commercial WFMS FlowMark [Leymann & Roller, 1994] and the research system Exotica [Mohan et al., 1995]. The *process activity* represent business actions; they may be composed of nested subactivities. An executing *resource* associated with an activity forms a *task*. *Data containers* persistently store input and output data of activities. The input containers of a process are mapped to different input parameters of process activities by variable definitions. *Data connectors* specify the data flow between process activities. *Control connectors* indicate the control flow and can be defined between process activities at the same nesting level of a process definition. *Transition conditions* are associated with each control connector and are evaluated after the activity preceding the control connector terminates successfully. *Exit conditions* express the conditions for successful termination of the activities.

3.3.2 Workflow Specification Languages

The different perspectives of a workflow metamodel must be described at the workflow specification level. One or more specification languages can be used for the expression of these perspectives. Most modeling approaches separate the specification of the workflow process (structure, behavior, and information) from the specification of the organizational and operational perspectives. The approach provides the advantage of separation of concerns making it possible to modify a process without having to change the organizational model (and vice-versa). Due to the limitations of some modeling formalisms, many approaches separate the informational (data flow) from the behavioral perspective (e.g., Mentor [Wodtke, 1997]). A detailed consideration and comparison of the various workflow specification languages is beyond the scope of our work. We consequently limit our discussion to a brief overview of common approaches, noting however, that different approaches are often equivalent with respect to their expressiveness and consequently can be mapped to each other.

Imperative Programming Languages

Procedural languages have been used for the expression of control (and data) flow in workflow systems. A disadvantage of these approaches is that they often lack theoretical foundations for the formal analysis and verification of the workflow specifications. This means that correctness of workflow specifications can only be expressed in connection with the implementation level, for example, in the language interpreter used for workflow execution. Furthermore, as no support is provided for the organizational perspective, these aspects must be defined in a separate formalisms which lies at a different level of abstraction.

In MOBILE [Jablonski & Bussler, 1996], for example, control is defined by control flow procedures which are sequences of special *control flow constructs*. ProcessWEAVER [Fernstrøm, 1993] provides an interpreted script language which provides built-in functions for list-processing, message passing, process data manipulation, and program invocation, and can be extended by libraries providing extended functionality (e.g., for document manipulation). A further imper-

ative process programming language is APPL/A [Sutton et al., 1995] which extends the programming language Ada with constructs appropriate for the specification of (software development) processes.

Graph-Based Approaches

Many systems use graph-based specification approaches. These are extensions of directed graphs, finite state automata or Petri nets. The extensions are mostly motivated by the need of expressing modular workflow hierarchies, data flow, and organizational aspects.

A large number of systems use various kinds of directed graphs to specify workflows. In general graph nodes represent workflow tasks and graph arcs represent the possible task execution dependencies. Systems which use directed graphs for workflow specification include FlowMark and Exotica [Mohan et al., 1995], ADEPT [Reichert & Dadam, 1998], ObjectFlow [Hsu & Kleissner, 1996], and TriGS_{flow} [Kappel et al., 1998].

Petri nets are particularly adapted to describe a concurrency-oriented workflow model by assuming a distributed state. They can be used as follows [van der Aalst, 1998]:

- *transitions* represent workflow tasks as well as possible outcomes of workflow tasks;
- *places* represent conditions which model the states between tasks (e.g., ready to start a particular task to which an arc is directed from that place);
- *tokens* represent the workflow cases (i.e., executing workflow instances).

An advantage of Petri nets is that they are amenable to formal analysis of execution semantics. This is however often not the case, when extensions of the basic formalism are introduced. In the commercial WFMS Leu [Graw & Gruhn, 1995], behavioral aspects are described with high-level Petri nets called FUNSOFT nets. INCOME/WF [Oberweis et al., 1997] uses nested relation transition nets where places represent structured objects and transitions are operations on its input and output places.

Workflow modeling with formalisms based on state machines has been pursued in a few research projects, prominent examples being Mentor [Wodtke, 1997] and METEOR [Krishnakumar & Sheth, 1995]. In Mentor, an extension of the *statechart* formalism [Harel et al., 1988] is used to represent workflow *execution states* and as a basis to define control flow. Statecharts are effectively finite state machines with transitions between states governed by ECA-rules.

Constraint-Based Approaches

Constraint-based workflow specification has its origin in AI techniques. The specifications are expressed with rules of some form (condition-action rules). In general, the condition specifies some predicate to be checked and the action represents the workflow task encapsulated by the rule. A *chaining policy* defines the control flow by determining when a rule automatically invokes other rules based on logical matching between them. In backward chaining, rules that may satisfy an unsatisfied condition are fired, while in forward chaining rules whose condition has been satisfied is fired. An example of the use of constraint-based workflow specification can be found in the class of systems based on the Marvel rule-based software process engine, i.e, Amber [Popovich, 1997] and Oz [Ben-Shaul & Kaiser, 1995].

3.3.3 Specification of Workflow Aspects with ECA-Rules

Due to the relevance to our work, we survey ECA-rule based workflow specification in somewhat more detail. ECA-rule based workflow implementation is considered elsewhere in this thesis. Note that ECA-rule based specification is also referred to as event and trigger-based modeling. The ECA-rule workflow specification approach has its foundations in ADBS technology. The ancestor of this work can be traced back to the seminal paper [Dayal et al., 1990] in which the use of triggers and database transactions is proposed for the specification of long-running activities. Various efforts for the further development of rule-based specification approaches appeared almost simultaneously. The concepts contributed by these approaches includes the use of ECA-rules for specification of control flow and the explicit definition of events at a semantic level capable of denoting situations of interest during workflow execution as well as time-related aspects such as deadlines, etc. Especially important in these approaches is the notion of event composition which allows the expression of complex workflow situations.

Bussler & Jablonski, 1994

In [Bussler & Jablonski, 1994] the specification of human agent notification and synchronization policies by ECA-rules is proposed. Tasks are entries in an agents's work-to-do-list; they are implemented by one or more operations which the agent can choose at any time. While all eligible agents are notified of a pending task, task execution has to be performed according to a synchronization policy (e.g., exactly once or fastest) enforced by ECA-rules. Events are primitive and refer to the updates of task status in the WFMS database. Conditions express integrity conditions on task/agent relationships (which agent updated the task status), agent properties (eligibility to execute a task), task completion status, and operation status. Actions update the task status in the database and agent worklists.

METEOR

In METEOR [Krishnakumar & Sheth, 1995], rules similar to ECA are used to express inter-task state and value dependencies. State dependencies specify how a controllable task transition depends from the observable state of other tasks. Events are implicit and signify the transition of a task to some state. Conditions are expressed over the output data values of tasks, global workflow variables, and filter functions which express the logical (application-level) success of a task. Actions enable the transition of a task to a new state.

SEAMAN

In our own previous work [Tombros et al., 1995], we advocate the use of ECA-rules for specifying the behavior of reactive components which execute cooperative processes. Specification is based on abstract and temporal events in an ADBS. Composite events express process-specific situations. Conditions are queries expressed over the event parameters and the persistent component state. Actions are synchronous and asynchronous requests exchanged between the components. Various elements of this approach are used in this thesis in which however, we consider ECA-rules as an execution-level mechanism (see part II).

WIDE

In the WIDE project [Ceri et al., 1997] ECA-rules are used for the specification of exception handling. Exceptions are raised by the occurrence of temporal, workflow-internal, and external events. Workflow-internal events refer to workflow variable updates, constraint violations, task cancellation or rejection, and unavailability of a processing entity. Conditions are expressed over

workflow variables or production data or can be temporal (an elapsed interval). Actions are notifications, task state modifications, or predefined exception handling routines.

Evaluation

ECA-rule based workflow specification provides the theoretical advantages of rich modeling functionality, some intrinsic support for workflow evolution, and the natural representation and handling of exceptional situations. However in practice, workflow specification with ECA-rules is low-level and tedious to use as it lacks support for the structuring of workflow definitions leading to “spaghetti specifications”. This has been recognized in the literature so that more structured modeling abstractions have been introduced in most cases. Thus in practically all considered approaches, ECA-rules are generated from a higher-level formalism and are used as an implementation mechanism (see below). There are however, hybrid modeling approaches, where ECA-rules are used to specify only selected aspects of the workflow metamodel (see Table 3-1).

Table 3-1: Use of ECA rules in workflow specifications.

<i>System</i>	<i>Rules</i>	<i>Use</i>
[Bussler & Jablonski, 1994]	ECA in relational ADBS	synchronization and notification of human actors
[Dayal et al., 1990]	ECA in object-oriented ADBS	activity ordering
METEOR	(E)CA	inter-task dependencies
SEAMAN	ECA in object-oriented DBS	activity ordering and actor notification
WIDE	ECA in relational ADBS	exception specification and handling

3.3.4 Transactional Workflows

WFMS which use as their underlying modeling paradigm *advanced transaction models* (ATM) view workflows as an extension of ATM. ATM extend the traditional ACID transaction model supported by DBMS to allow advanced application functionality (e.g., with respect to task collaboration) and improve performance (e.g., by reducing transaction blocking). The following are the most important extensions to the classic ACID transaction model [Elmagarmid, 1992]:

- *Nested transactions* extend the single-level transaction structure to multi-level structures. They provide full isolation at the top-level transaction level but provide improved transaction program modularity, finer granularity of failure handling, and increased inter-transaction concurrency. *Open nested transactions* relax the isolation requirements of nested transaction by making the results of subtransactions visible to other concurrently executing subtransactions.
- *Sagas* consist of a set of ACID transactions with a predefined order of execution and a set of *compensating subtransactions* which are executed if one of the saga subtransaction fails. Sagas relax the isolation requirements and increase inter-transaction concurrency.
- *Multi-level transactions* combine nested transactions with compensating subtransactions allowing subtransactions to commit before the top-level transaction. Their results however are visible only to subtransactions which commute the committed ones. In case the global transaction aborts the effects of committed subtransactions are undone by the compensating subtransactions.

- *Flexible transactions* are sets of tasks with a set of functionally equivalent subtransactions for each and a set of execution dependencies on the subtransactions. Isolation requirements are relaxed by use of compensation; atomicity requirements are relaxed by the specification of valid termination states in which some of the subtransactions may be aborted.

In general, the ATM-based approaches introduce in the representational domain of workflows the notion of workflow transactions. These are not equivalent to database transactions but allow the enforcement of relaxed transaction semantics to a set of tasks. Tasks are viewed as transactions or black boxes whose internal sequential processing details of operation are not important for the workflow system, whose “correctness” however has to be ensured. This means concretely that a workflow transaction should ensure consistency from a business perspective. In other words, a workflow transaction is a sequence of workflow tasks which transfers a business process from one consistent state into the next consistent state. Only some functional aspects of the task are externally visible: some of its execution states, legal transitions between these states, and the conditions that enable these transitions.

While commercial workflow management systems are not based on transactional notions, many research efforts have been made in this direction. In [Dayal et al., 1990] workflows are seen as long-running activities which may have a recursive structure. Control flow may be defined either with ECA-rules or in the activity’s description (see below). Consequently the, use of workflow transactions permits the construction of a hierarchical view of a business process by nesting workflow transactions over multiple levels [Chen & Dayal, 1996].

ConTracts [Wächter & Reuter, 1992] are a transaction-grouping mechanism which provides relaxed atomicity and relaxed isolation properties. They consist of predefined actions called steps and control flow defined in scripts. Contracts are forward recoverable.

[Georgakopoulos et al., 1994] define an ATM framework in which workflows can be defined consisting of constituent transactions corresponding to workflow tasks. The workflow structure and further correctness criteria are defined by the ATM. The Distributed Object Management System [Georgakopoulos & Hornick, 1994] implements these concepts.

Evaluation

The most important contribution of transactional workflow modeling are the concepts they provide for the expression of correctness criteria for workflows based on the formal work done in the context of ATM. However, despite the contributions of ATM to workflow modeling, it has been recognized, that they provide too limited modeling features and can be mainly of a supportive nature in workflows. For a normative comparison of ATM and workflow systems we refer to [Worah & Sheth, 1997].

3.4 Architecture Styles for Workflow Management Systems

We evaluate the two principal attempts to define a reference architecture for workflow management systems: the Workflow Management Coalition Reference Architecture and the Mercurius project. We subsequently discuss various architectural styles for workflow systems by examining research prototypes and commercial systems.

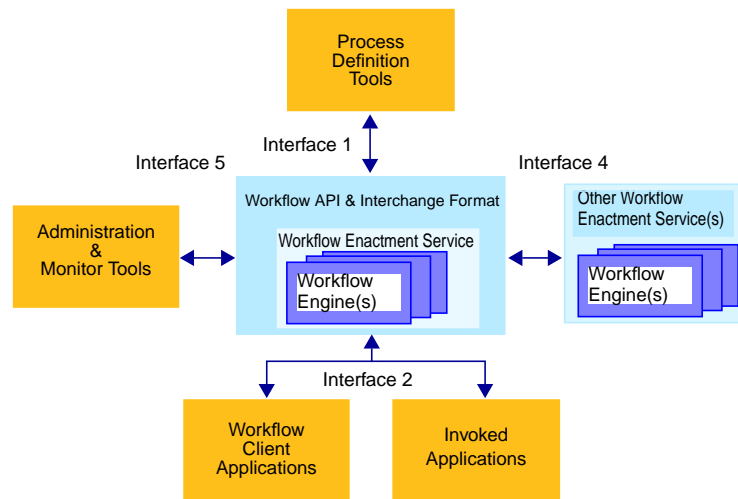


Figure 3-3: The WfMC Reference Architecture (adapted from [WfMC, 1994]).

3.4.1 The WfMC Reference Architecture

The WfMC was established in August 1993 as a non-profit international body for the development and promotion of workflow standards. The WfMC has defined the Workflow Process Definition Metamodel [WfMC, 1998] described in section 3.3.1, and an architecture reference model [WfMC, 1994] describing the functional constituents of workflow management, and information flows between them. In this section we briefly present the basic concepts and structure of the reference model, and discuss the contributions of such a model towards the description of operational aspects of workflow systems and workflow system composition. For a more detailed description we refer to the WfMC documents.

Functional Areas

According to the WfMC the term *workflow* refers to the computerized facilitation or automation of a business process [WfMC, 1994]. The automation is defined within a *process definition* which identifies the activities, procedural rules and control data used to manage the workflow during its *enactment*. During the enactment, documents, information, or tasks are passed among the various participants. WFMS are systems that allow the definition, creation, and management of the execution of workflows through the software which is driven by a computer representation of the workflow logic. The reference architecture defines the following WFMS functional areas:

- *Build-time functions* concerned with defining and modeling the workflow process and its constituent activities: these functions result in a computerized definition of a business process. During the definition phase, a business process is translated from the real world counterpart —defined in terms of business domain elements— into a formal, computer processible definition by the use of one or more analysis, modeling and system definition techniques.
- *Run-time process control functions* concerned with managing the workflow processes in an operational environment and sequencing the various activities to be handled as part of each process. At run-time the process definition is interpreted by the *workflow engine*, which is responsible for creating and controlling operational instances of the process, scheduling the various activities steps within the process and invoking the appropriate human and IT application resources, etc.

- *Run-time interactions* with humans (e.g., form filling) and IT application tools (e.g., an client of an order database) for processing the various activity steps. Interaction with the process control software is necessary to transfer control between activities, to ascertain the operational status of processes, to invoke application tools and pass the appropriate data, etc.
- The ability to *distribute tasks and information* between participants is a major distinguishing feature of workflow run-time infrastructure. According to the WfMC, the distribution function may operate at a variety of levels (workgroup to inter-organization) depending upon the scope of the workflows; it may use a variety of underlying communications mechanisms (electronic mail, messaging passing, distributed object technology, etc.).

WfMC-Interfaces

The WfMC Reference Architecture distinguishes six different functional subsystems of a WFMS and five groups of interfaces (see Figure 3-3). In the current standard these interfaces are specified as C(-level) language function signatures with input and output parameters as well as a return type, and include (as of April 1998). The subsystems distinguished are the following:

- The *workflow enactment subsystem* provides workflow enactment services by one or more workflow engines.
- The *process definition tools* implement the previously mentioned build-time functions. They interact with the workflow enactment subsystem through *interface 1* [WfMC, 1998] which defines a simple API for the interchange of workflow specification elements. The underlying workflow specification schema is the WfMC workflow metamodel.
- *Workflow client applications* are workflow enabled applications which provide process and activity control functions, as well as worklist management, and administration functions. *Interface 2* [WfMC, 1996a] defines an API to support interaction with the workflow client applications. The WfMC mentions different possible configurations which essentially refer to the component which administers the worklist.
- *Invoked applications* execute workflow tasks. While initially an *interface 3* was announced which would support interaction with invoked applications, it has subsequently been amalgamated in interface 2.
- *Other workflow enactment services* are accessed by *interface 4* [WfMC, 1996b]. It defines an API to support interoperability between workflow engines of different vendors allowing the implementation of nested subprocesses across multiple workflow engines.
- *System monitoring and administration tools* interact with the workflow enactment subsystem through *interface 5* [WfMC, 1996d]. The interface defines an API for the administration of system monitoring and audit functions for process and activity instances, remote operations, as well as process definitions.

Evaluation

The WfMC Reference Architecture has been developed abstracting from a concrete WFMS by identifying the interfaces which enable subsystems to interoperate through specific access points. The architecture identifies the major functional components and is essentially a collection of programming language-level descriptions of the main subsystem interfaces. The goals and the historical conditions underlying the development of the model, as for example, the fact that it is a ven-

dor consortium, have led to rather unsatisfactory results. The main critique can be summarized as follows:

- The WfMC Reference Architecture is useful as a discussion basis to identify the high-level structure of a workflow system. However, the components one might expect to encounter (e.g., a workflow engine or invoked applications) are only described through their interfaces. At this level of abstraction, no useful operational description of a workflow system can be made; the nature and *modus operandi* of the different components are not further described.
- The level of detail provided by the WfMC Reference Architecture is too abstract to be used in the design and development of the individual constituent components. It does not provide a development framework or a composition method for WFMS.
- The interfaces defined are minimal as they tend to mirror the least common denominator of functionality present in the vendor products. The interfaces have been defined at a low level (in the C programming language).

3.4.2 The Mercurius Reference Architecture

Recently, an effort to define a comprehensive reference architecture for WFMS has been made [Grefen & de Vries, 1998]. It represents the results of a collaboration project between the academic community and industry in The Netherlands. The goal of this effort has been the identification of all WFMS components and their functionality. According to the reference architecture defined, a WFMS consists of three modules:

- the *design module* which provides design services for workflow applications;
- the *server module* which provides central workflow enactment services; and
- *workflow clients* which provide decentralized end user services for workflow enactment and management.

Other identified modules include a communication system for information exchange between workflow servers, application systems providing business-specific functionality, and a DBMS which stores workflow definition and workflow enactment data.

An interesting aspect of this architecture is the provision for extensibility. The basic functionality of the WFMS can be extended through a “software bus” to which *extension modules* can be attached. These extension modules can provide additional design or enactment functionality. Examples of enactment extension modules that are mentioned include a resource allocator and an exception handler. The approach is similar to our proposal for extending the basic workflow engine functionality with extender EOB (see chapter 6). It is based however, on a different integration paradigm.

Despite the considerable detail of the description of a WFMS in [Grefen & de Vries, 1998], the value of the contribution with respect to defining a reference architecture is restricted due to the following:

- the kinds of relations between the various modules are undefined;
- the process and coordination structure (i.e., the dynamic aspects of the architecture) are not defined;
- the conceptual data model underlying the architecture is not discussed; and

- no attempt is made to provide a development perspective and some guidance for the composition of WFMS.

3.5 Workflow Application Metamodels

Workflow metamodels may provide an abstract view of workflow applications, i.e., an application metamodel. Workflow applications are logical units which can be implemented by different software components (usually some kind of executable program). These programs usually differ with respect to their call properties, call formats they understand, etc.

The two reference models we considered in the previous section distinguish between a design or build-time environment and a run-time or execution environment. With respect to components in the run-time environment, the WfMC workflow metamodel [WfMC, 1998] only distinguishes between workflow aware applications and invoked applications. Principal examples of workflow aware applications include worklist managers (providing the system interface for end-user interaction), and administrative tools.

In the rest of this section, some examples of metamodels of workflow applications are described. As practically all commercial WFMS provide only rudimentary application modeling limited to specifying the external application call and its parameters (e.g., [COSA, 1998, CSE, 1996, FileNet, 1998]), we concentrate here on various research prototypes.

Mobile

In [Jablonski et al., 1997], a generic workflow application metamodel is described which abstracts from application implementation details (see Figure 3-4). A workflow application is an implementor of a workflow task. A workflow application is associated with a set of application programs (called a program group) which are equivalent with respect to the “goal attainment” of the workflow but differ in their functionality and call parameters. Application programs are distinguished into simple program calls or extended programs which consist of further steps beside the program execution. In that sense, an application wrapper can be considered an extended program. The suitability and successive selection of the program group for a workflow task can be derived by the program group’s functionality. Not all programs in a program group provide the same functionality so that the most appropriate program is either statically assigned to a workflow task (during workflow specification) or dynamically determined. The authors propose that by successively defining call or other properties, an application hierarchy can be created which may be the basis of a class hierarchy. Among the definable application properties we mention the following:

- the type and name of the application;
- the way in which the application is called at an operating system level; this can be a program execution, an API call, or an SQL statement;

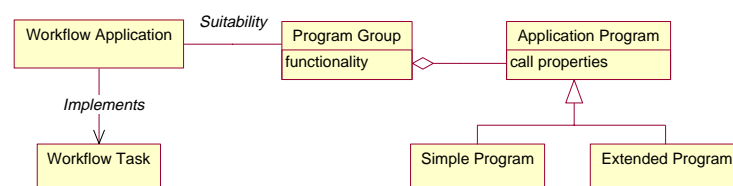


Figure 3-4: The workflow application metamodel proposed in [Jablonski et al., 1997].

- transactional/non-transactional application; and
- the input and output data for the application.

Thus each application is described by the calling mechanism used to initiate its execution and the mechanism used for parameter passing which is handled by a wrapper. Applications may be arbitrarily distributed as can user agents. In order to application location distribution proxy mechanisms have to be used [Shapiro, 1986]. Workflow applications can be subsequently called from workflow type specifications written by the MOBILE scripting language. This interesting approach however is not followed by an operational classification of program properties.

METEOR and Mentor

In Mentor [Wodtke, 1997] and METEOR [Krishnakumar & Sheth, 1995], external applications are considered as CORBA objects and consequently modeled by IDL specifications of wrapper objects. The communication with the external application is implemented through an ORB. As an advantage of this approach, the Dynamic Invocation Interface of the ORB can be used for late binding of application objects. The main disadvantage lies with the assumption that a CORBA compliant wrapper must be developed for each participating application.

WIDE

In WIDE [Ceri et al., 1997] agents are explicitly defined as part of the organizational structure through which a workflow is executed. Various subtypes of agents are distinguished. Individual agents are either automated or human and are described by attributes including the agent name, its network domain, and its availability. External applications are called based on attributes of data flow elements. A document file for example has as its attributes the operations `edit` and `print` which are strings containing program calls at the operating system level. Various special agents have to be defined in relation to a workflow instance (instance executor, instance responsible, and task executor) and a workflow specification (the workflow designer and the workflow domain administrator).

ObjectFlow

ObjectFlow [Hsu & Kleissner, 1996] provides concepts for the explicit description of processing entities comprising a workflow system. In general, it defines a generic workflow system architecture which distinguishes between three different types of components:

- The *flow controller* provides services for defining, creating, executing, and monitoring workflows. It is the actual workflow engine.
- The *flow agents* are customizable components which act as clients to the flow controller services. They serve as integration components between end-user tools and the flow controller. Two classes of flow agents are distinguished: worklist agents provide work notification to resources; application agents provide a task execution environment and are invoked by a resource to execute an atomic activity.
- *Service agents* can be used to extend the services provided by the flow controller. Two different service agents are mentioned: a policy resolution agent which provides services for specifying and enforcing rules by which authorized resources are associated with workflow activities; a distribution service agent allows multiple flow controllers to interoperate.

The main strength of the approach lies in the conceptual identification of components comprising the workflow system. Furthermore, the base functionality of the system can be extended by implementing additional service agents. It is proposed that the various kinds of agents can be implemented as CORBA objects in which case the distribution of the system can be directly achieved. No further information on agent specification and implementation was available however at the time of this writing.

CapBasED-AMS

In CapBasED-AMS (Capability-based and Event-Driven Activity Management System) [Karlapalem et al., 1995], workflow tasks are executed by problem solving agents (PSA) representing processing entities with specific task execution *abilities*. PSA can be either active (humans) or passive (applications). PSA descriptions include a list of roles taken by the PSA, the location of the PSA, and a list of constraints. Roles have a 1:n relationship with PSA. They are used to store the PSA ability tokens. PSA role descriptions comprise a unique role name, a description, a type (active or passive), a list of ability tokens, and a list of constraints on PSA with this role.

Activity *need* specification takes place separately from PSA ability specification. PSA selection for task execution is based on matching of the task needs with the PSA abilities. Dependencies between tasks are defined as composite events with similar semantics to those defined in the SNOOP event algebra [Chakravarthy & Misra, 1994]. Task completion is signalled by events generated by the PSA corresponding to successful execution or failure, however, additional event types may be specified to convey the execution status of a task.

TriGS_{flow}

TriGS_{flow} uses a rule-based approach for agent coordination [Kappel et al., 1998]. Activities are executed by agents selected at run-time, based on agent-role relationships or workflow data. The selection returns a single agent. Agents are modeled by objects from a type hierarchy distinguishing between automatic and human agents. They are implemented as independent processes communicating by messages. Human agents manually select an activity to be started and either execute it manually or interactively with an application. External applications are started by a shell command of the underlying operating system. External applications are integrated through shell commands. Internal applications are objects of the underlying object-oriented database system and can perform various processing activities in the system. The modeling of these object provides a mechanism for the extensibility of the basic workflow system architecture. The specification of internal applications objects is not described however, in more detail in the available literature. Presumably some OODB application design and development method can be used (e.g., [Embley, 1998]) although it not clear how this is integrated with workflow system development.

Oz

In Oz [Valetto & Kaiser, 1996] the issue of tool representation in a process-centered software engineering environment has been researched. The concept of tool integration *envelopes*, introduced in [Dowson, 1987], is extended for the integration of interactive, multi-user applications. Envelopes realize the so-called “black-box integration” where the granularity of integration is restricted to call/return semantics involving application invocation, application execution, and subsequent result evaluation. Envelopes handle interfacing with non-interactive applications by providing the following functionality:

- They invoke an instance of the application when necessary. They parameterize the instance according to the workflow task. The parametrization may refer to call options or preparation of corresponding input data.
- They transform data from the workflow system representation to/from the representation understood and produced by the application.

In Oz, a protocol is defined which allows the submission of multiple activities to the same executing tool instance. Orthogonally, multiple users may interact with a tool following this protocol. A tool declaration in Oz contains the following information:

- The protocol for activity submission to the tool: a tool can be either multi- or single-user, depending on if multiple users can share the same invocation instance (the former). Orthogonally, a tool instance may allow or disallow concurrent (overlapping) execution of multiple activities.
- Invocation information including the path in the file system where the tool (or its envelope) resides, an optional host address where the tool must execute, the operating system on which the tool is expected to run, and the maximum number of copies of the tool that can execute at the same time.

Evaluation

We summarize our evaluation of application metamodels in workflow systems as follows:

- We conclude from our survey that with the exception of Mobile, most systems attempt to abstract completely from implementation details at the modeling level. Although this is desirable for the modeling of, say, control flow, important features and properties of applications may be hidden. Such features are essential for the thorough understanding of issues which come into consideration when selecting an application to implement a specific task. Most metamodels provide only rudimentary concepts for the reuse and integration-oriented characterization of workflow applications.
- Furthermore, the extensibility of WFMS kernel functionality is not supported by most metamodels. There is no way to describe functional components of the workflow infrastructure, i.e., the workflow client applications and the workflow engine itself. Thus the workflow system architecture is not described explicitly and consequently it cannot be extended in a controlled way; furthermore, reasoning about WFMS properties at an architectural level is not supported.

3.6 Summary

In this chapter we presented the domain of discourse of workflow development in the form of workflow metamodels and the conceptual tools which are used for describing workflow models, i.e., the workflow specification approaches. We also described two existing efforts to describe both conceptual and architectural aspects of workflow systems. We have seen that these models do not provide sufficient conceptual and methodological support for the design and implementation of operational workflow systems. In that sense, they cannot be considered to be complete domain specific software architectures according to the meaning of the term defined in chapter 2.

We concluded this chapter by surveying workflow application metamodels in workflow management systems. It is obvious from the survey that almost no support for consideration of workflow system components at an architectural level is currently provided. This issue is considered in the next chapter where we analyze workflow system architecture.

Part II: Workflow System Architecture

The second part of this thesis introduces appropriate abstractions for the specification of workflow system architecture as provided by the REWORK environment. These abstractions are provided by the workflow system architecture metamodel developed as part of this thesis which proposes an component and event-based style for the specification of workflow system architecture. The metamodel provides a uniform specification, implementation, and integration framework for workflow application systems and the workflow management systems. The underlying assumption is that every individual subsystem of a workflow system and thus ultimately the entire workflow system can be composed out of *reactive components* which use an event-based coordination infrastructure. This metamodel is called the REWORK metamodel.

In general, the abstractions defined in the REWORK metamodel are available to the workflow system composer by the REWORK environment. This environment provides access to the constructs implementing these abstractions by a REWORK metamodel-compliant specification language (textual or graphical). With the help of these constructs, the workflow system composer can define workflow system architectures henceforth called REWORK systems. The elements of these architectures are stored in a build-time repository which is part of the REWORK environment.

Before introducing the elements of the REWORK metamodel, we provide in chapter 4, an analysis of the architecture of workflow application systems and workflow management infrastructure elements. The results of this analysis can be expressed in a systematic way by the introduction of *actor connotations*. These connotations provide the conceptual framework for the characterization of workflow system actors and the subsequent selection of their implementation components in the REWORK environment.

In the subsequent three chapters we describe the various aspects of the REWORK metamodel. In chapter 5 we define the connector types provided by the REWORK metamodel. In chapter 6 we describe the types of components and provide formal semantics for workflow execution by these components. In chapter 7 we describe the mapping of workflow specifications to a workflow system architecture and describe an example of a seamless extension to the kernel WFMS functionality.

4 Analysis of Workflow System Integration Architecture

In this chapter some aspects of the conceptual architecture of workflow systems defined in the REWORK metamodel are analyzed in detail. Based on two workflow application examples, we describe the issues relevant to the nature and integration of actors in a workflow system. Our interest, however, is not limited to the implementation of workflow applications. In REWORK, workflow system architecture comprises both workflow applications and the workflow management infrastructure, i.e., WFMS kernel and extensions. Consequently, we survey the types of actors encountered in workflow systems. We subsequently propose the extensible classification scheme for the various actor types provided by the REWORK metamodel which focuses on the workflow integration perspective. This perspective considers which kinds of interactions a specific actor supports with its environment and how these affect workflow execution and workflow system implementation. The main purpose of the classification is analysis and documentation of the workflow system that is being composed. Based on this analysis, various *integration component templates* can be selected and customized for a specific actor during workflow system composition; these are subsequently used in a plug-and-operate fashion to compose an operational workflow system.

4.1 From Actors to Integration Components

Components and *componentware* are recent buzzwords in information technology, especially in relation to Internet technology and the Java programming language. The notion of a component is somewhat arbitrarily defined in the literature and often depends on the perspective of the author.

From a programming language perspective, components are considered as data encapsulation units with a clearly defined data manipulation interface [Pree, 1997]. They correspond to the compilation units of the programming language. For example, they can be Modula-2 modules, or C++ and Java class instances. The granularity of components may be different depending on the kind of system that is composed. There is consensus however, that components are large-granularity entities. A design pattern [Gamma et al., 1995] is a reusable collection of objects or object classes that cooperate to achieve a design goal. It encompasses knowledge expertise for the solution to a recurring well-defined problem [Bass et al., 1998]. A component may be implemented by means of one or more design patterns.

From a methodological perspective, components are characterized by the fact that they have been designed to be (re)used in a compositional way as part of a framework of collaborating components [Nierstrasz & Dami, 1995]. The property of information hiding and complete specification through a component's interface is important for the reusability of a component. If the implementation of a component is completely transparent and composition of components is based solely on the specification of the component's interface, then the component can be replaced by an improved version, or by a completely different implementation, without affecting the rest of the system. This property is essential for the composition of large complex systems.

From a software architecture perspective, components are the composition units of a system [Shaw & Garlan, 1996]. They are the loci of computation and state, they have an interface specification that completely defines their properties and a specific type. Examples of component types include filters, servers, and memory. The named entities visible in the component interface are its interface points. This definition actually expands the notion of a component as an abstract data type. Components may be primitive or composite in which case they define configurations.

System architecture, however, cannot be specified solely based on component interfaces. A further aspect has to be considered: the outgoing interface and component connectors. The *outgoing interface* refers to the interfaces on which a component relies, i.e., the interfaces of other components used by the original component. The outgoing interface of a component is part of the properties of the interface as used by the component itself. In languages such as Modula-2 or C such relationships are inadequately captured by import or includes clauses. These relationships must also be made explicit. *Connectors* explicitly describe the interactions between components, they are the loci of relations among components [Shaw & Garlan, 1996]. They have defined properties with respect to the types of interfaces they mediate for and the kind of interaction that they provide. They have a specific type and define roles of the components they connect. Examples of connectors include events, remote procedure calls, and pipes.

Actors include the application software and the representations of human beings which participate in workflow execution. Components and connectors are implementation level constructs which define the concrete structure and communication paths of the software comprising a workflow system. In other words, an actor exists independently of its integration and representation in a concrete workflow system. In the REWORK metamodel we explicitly consider the integration-related properties of an actor: these properties comprise the *actor connotation*. The actor connotation associates the relevant properties of the actor with the characteristics of the components and connectors required to adequately represent the actor in the workflow system.

In order to clarify the notion of actor connotations, we define two (human) roles in the workflow system development process, i.e., users of the REWORK metamodel:

- The *workflow system architect* characterizes various types of actors, with respect to their integration properties, with the help of entity connotations. The resulting classification is the basis for the implementation of standardized component templates stored in a development repository and with specific properties derived from the actor connotations. These component templates can be reused in various workflow systems.
- The *workflow system composer* characterizes the actors required for the implementation of tasks defined in a workflow specification. The characterization serves in selecting the appropriate component templates which are then customized to implement the actors of the composed workflow system. Once these components have been customized and stored in a run-time repository, the resulting workflow system can be put into operation.

In other words, the systematic characterization of actors by the workflow system architect allows the workflow system composer to subsequently integrate actors and thus compose the workflow system by customizing a standard set of components and connectors for the particular actor. The mapping from actors to composite executable implementation components is a built-in functionality of the workflow system composition environment.

Workflow specification can be based on connotations of existing actors but also identification of new ones. In all cases, it implies the customizing of components it uses for the purposes of particular workflow types. The relations between various workflow system composition elements in the REWORK metamodel is depicted in Figure 4-1.

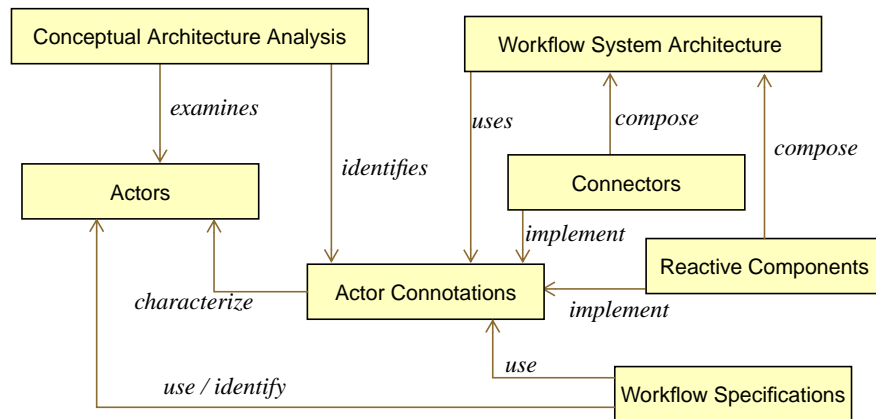


Figure 4-1: Workflow system composition elements as defined in the REWORK metamodel.

In the next sections, we discuss the kinds of actors encountered in workflow management applications by surveying two case studies from the industry.

4.2 Examples of Workflow Application Systems

In this section we consider two typical examples of workflow application systems. The first application is representative of the use of workflow management technology in an industrial design setting, while the second is a typical document routing and forwarding application. We note that the heterogeneity of integrated applications is usually a matter of fact in commercial settings as the exchange of a given application is often not a feasible option not only due to technical but also due to financial or human-related factors, as for example, user acceptance.

4.2.1 The Product Design Change Workflow

Survey Co¹ is a worldwide leading manufacturer of surveying instruments. Its product line also includes photogrammetry systems, GPS instruments, and industrial measuring equipment. We consider the *product redesign workflow* (PRW) in the engineering department. The PRW is characterized by various participating organizational departments which have to make decisions referring to a particular change to a product design document proposed by an engineer. The participating departments include the engineering department where the change is requested, the production department where the effects of the change on actual and future production plans and stocks is considered, and the cost accounting department where the effects of the change on modifications to existing goods in production and in warehouses is calculated. The proposals and information of these departments are considered in order to decide if and under what terms the product design change is accepted.

The PRW is business-critical and is executed hundreds of times a month. The normal duration lies between four and ten working days, although expedient execution is sometimes requested for particular changes, especially as production of the changed parts/products comes to a halt during a large portion of the change process. The PRW includes various decision making and interactive activities which are performed by people or groups of people based on the provided information. These include the acceptance or rejection of the change by a change control group and the formulation of an alternative redesign request by the engineer in case the original request is rejected.

¹ The company name is fictitious.

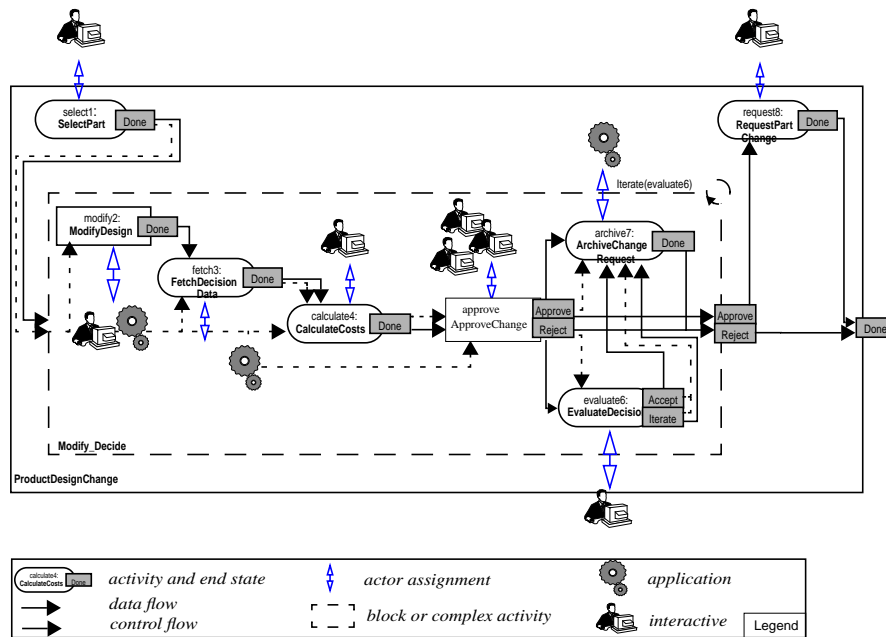


Figure 4-2: The product redesign workflow in the TRAMs notation [Kradolfer & Geppert, 1999].

Often, the determination of the members of the various groups which have to make a decision is dynamically determined based on workflow production data such as the concrete product being changed, the scope of the change, etc. Finally, we note that the workflow outcome has an effect on other processes in the company such as the planning of production and stock maintenance for modified products and parts.

Table 4-1: Applications used in the Survey Co product redesign workflow.

Application	Functionality	Used interfaces
CADIM/EDB ^a	design artifact management	proprietary DML
Pro/Engineer 2D ^b	CAD	application invocation, GUI
SAP R/3 ^c	OLTP	RPC-based API, SQL
DZA-Viewer	design document display, "redlining"	application invocation, GUI
Microsoft Word, Microsoft Excel ^d	document editing	application invocation, OLE/COM, GUI
Microsoft Mail	electronic mail	MAPI, GUI
Keyfile ^e	business document archival	proprietary API, SQL for document retrieval, GUI

a. CADIM/EDB is a product of EIGNER + PARTNER.

b. Pro/Engineer is a product of RAND TECHNOLOGIES GmbH

c. SAP R/3 is a product of SAP AG

d. Microsoft Word, Excel, and Mail are products of Microsoft Corporation

e. Keyfile is a product of Keyfile Corporation

The PRW begins when an employee creates an error or change requirement report for a product part. An engineer responsible for this part receives this report makes and prepares a request for a change in the design of the part. He subsequently uses a CAD application to modify the part which is stored in an engineering database system. Once a new version of the redesigned part is created the change request has to be evaluated from a cost perspective. Existing and planned or-

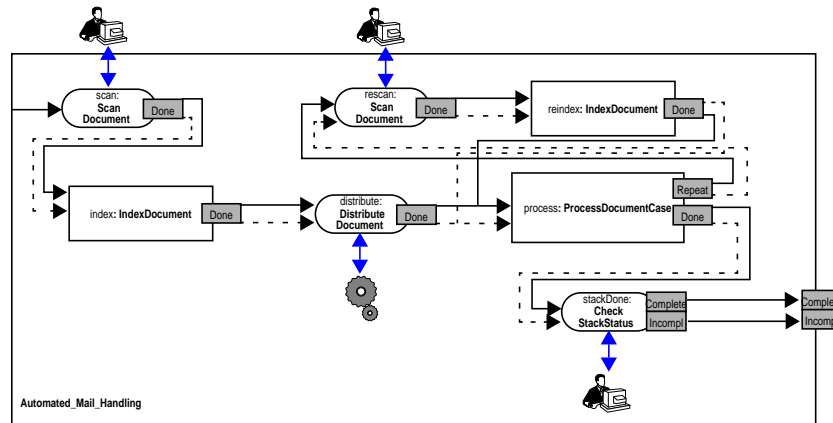


Figure 4-3: The mail handling workflow in the TRAMs notation [Kradolfer & Geppert, 1999].

ders for the part, stored quantities, material numbers for subparts, have to be considered by an accountant who calculates the cost of implementing the change in the part design. The calculated costs are the basis on which management decides if the requested change is feasible.

The decision is a complex group activity in itself, in which various additional factors have to be taken into consideration. A positive decision leads to the implementation of the change, while a rejection may be evaluated by the engineer who originally proposed the change who may subsequently decide to resubmit a design proposal leading to a new iteration of the whole modification and decision process. Independently from the decision, the documentation resulting from the current modification/decision cycle is archived. The workflow is depicted in Figure 4-2.

The historical development of Survey Co has lead to a heterogeneous software landscape in which various existing applications have to be integrated in the workflow system. These include CAD software, an engineering database system, a business application suite, a document management system, an e-mail system, and standard office applications. The workflow applications and integration interfaces are described in Table 4-1.

4.2.2 The Automatic Mail Handling Workflow

IAA¹ is a medium-size health insurance company. The second case study refers to the efficient and automated handling of certain types of incoming mail at a large insurance company. An internal study at IAA crystallized the requirements for an optimized handling of incoming mail. Mail handling takes place at three regional service centers which from the perspective of the mail workflow are divided in a group responsible for incoming mail handling and distribution, and in groups which are responsible for performing case processing and management, invoice control, customer information, and archival of cases based on geographical criteria. The types of documents that should be handled by the system include invoices, accident reports, cost estimations, and various correspondence. The system must support an optimized handling of the standardized correspondence, that is the three first mail categories which are based on predefined forms.

The mail handling workflow is considered from the perspective of a particular document flowing through the system. An incoming letter is opened and added to a pending mail stack which is subsequently scanned in its entirety. Document images with an identifiable barcode, encoding an insurance policy number, are automatically indexed based on customer data retrieved from the company database. If the barcode is unreadable or the barcode sticker is missing, the

¹ The company name is fictitious.

document image has to be manually indexed. Manual indexing is performed by a member of a clerk pool who retrieves the necessary index data through predefined queries. Once the indexing form is filled, indexing data is attached to the document image as in the automatic case. After indexing is concluded, the document—depending on its type and the indexing data—is forwarded to a group worklist. Members of the group select a document from the worklist to process. This is done interactively and consists of data retrieval and form filling activities as well as invoice control. Once case processing is completed, the document image and the index data are archived. In case a document image is illegible, the staff member may request that the document is re-scanned and re-indexed. Once all documents in a particular stack have been successfully processed, the stack can be destroyed. The high-level structure of the workflow is depicted in Figure 4-3.

The workflow system in this case study had to be integrated in a dynamic environment which was undergoing step-wise re-implementation. The newly composed workflow system, in addition to integrating various legacy applications operational on a variety of operating system platforms and database systems, should provide new functionality. Furthermore, it should be extensible in the sense of being able to be interoperate with CORBA-based applications and wrappers as these are being developed. The resulting workflow system should allow distributed workflow execution in three sites connected by a company-owned network. The workflow applications and integration interfaces are described in Table 4-2.

Table 4-2: Workflow applications in the IAA workflow system.

<i>Application/Platform</i>	<i>Functionality</i>	<i>Used interfaces</i>
n.a. / Windows NT	document scanning	GUI, application invocation, OLE/COM
n.a. / Windows NT	document management	GUI, OLE/COM for document storage, SQL for document retrieval
IS applications / MVS	case processing	GUI, CICS
TeraData ^a / NCR Unix	archival	ODBC
SAP R/3 ^b / Solaris	case processing	batch file input
HAI / Solaris	invoice control	CORBA IDL
Lotus Notes / Windows NT	document forwarding, electronic mail	GUI, Lotus script language

a. TeraData is a product of NCR Corporation

b. SAP R/3 is a product of SAP AG

4.2.3 Evaluation

The two case studies represent typical examples of the heterogeneous nature of workflow application systems. The heterogeneity of the involved systems spans operating systems, middleware, provided interfaces, application types, and use scenarios. It is obvious that the task of the system integrator is far from trivial, especially if the developed system is to remain extensible and open. The weakest possible technological assumptions should be made and the dependencies between the integrated systems should be confined to the workflow system, i.e., existing applications should remain unaffected by the workflow operation; furthermore they should be exchangeable if this is required without disturbing the workflow system operation.








4.3 Implementation Architecture of WFMS

In the previous chapter we described two proposed reference architectures for WFMS: the WfMC reference model [WfMC, 1994] and the Mercurius reference architecture [Grefen & de Vries, 1998]. The main contribution of the presented reference architectures is the identification of functional domains of WFMS and the relationships between these domains. The kernel domains identified exist in every WFMS independent of its implementation. Consequently, functional domains do not have to necessarily correspond to architectural subsystems or WFMS components. The mapping of these domains to specific components and connectors is the task of the workflow system architect.

In this section we turn our attention to the implementation architecture of WFMS, i.e., we consider how the identified functional domains are mapped to WFMS software components and subsystems and what kinds of interactions occur between these subsystems. We consider a series of commercial systems and research prototypes emphasizing on the distribution and application integration aspects of the architecture. The considered WFMS are classified along the distribution dimension of the *WFMS server subsystems*. We can distinguish between the following architectural variations:

- *Centralized server*. The WFMS is built around a centralized server subsystem, for example a centralized DB server. Two possible configurations can be distinguished:
 - one workflow enactment server exists in the system, or
 - multiple workflow enactment servers share the centralized component.
- *Replicated servers*. The WFMS is based on multiple distributed server subsystems which are identical among them and serve all workflow clients.
- *Localized servers*. The server subsystems are localized from the perspective of the users or the applications which execute workflow activities.
- *No servers*. The workflow system is fully distributed and does not have identifiable server subsystems. Server functionality is implemented by the workflow clients.

In the next subsections we consider both commercial systems and research prototypes and discuss the advantages and disadvantages of each approach. For two more general surveys concerning various aspects of commercial WFMS we refer to [Alonso & Schek, 1996] and [Alonso et al., 1997a]. In the architectural diagrams used throughout this thesis, the following notational conventions are used:

- Identifiable WFMS components are depicted by  with the name of the component written inside.
- Components external to the WFMS which are however necessary for its operation are depicted by  with the name of the component written inside.
- Runtime objects are depicted by 
- Organizational units and functional subsystem divisions which do not correspond to components are depicted by  with the name of the subsystem written inside.
- Interactive client applications are depicted by 
- Invoked applications are depicted by 
- Components providing persistent storage functionality are depicted by 

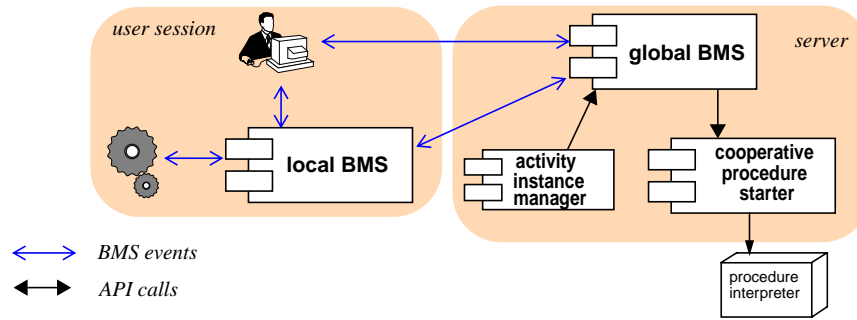


Figure 4-4: The implementation architecture of Process WEAVER-based workflow systems.

- Connectors are depicted by different kinds of arrows. The type of the connector is described in the diagram. The arrow is directed away from the initiator of the interaction.

4.3.1 Centralized Servers

Various commercial systems such as Process WEAVER [Fernstrøm, 1993], ProMinanD [Karbe, 1994], WorkParty, and Staffware have a centralized architecture. They are built around a centralized workflow enactment server and a centralized data storage system which is accessed for the execution of every workflow step. The principal disadvantage of centralized systems is the lack of scalability to support large-scale distributed workflow execution.

Examples of research proposals which follow a centralized architecture include ObjectFlow [Hsu & Kleissner, 1996] and most systems which are built around an active DBMS used for workflow enactment, such as the Panta Rhei system [Eder & Groiss, 1996] and TriGS_{flow} [Kappel et al., 1995]. We limit our discussion to two systems in this section based on their relevance to our work: Process WEAVER and TriGS_{flow}.

Process WEAVER

Process WEAVER [Fernstrøm, 1993] originally developed as a process enactment engine for software engineering environments was subsequently marketed by Cap Gemini Innovation as a workflow management system. The information described here is based on the system administration manual of Process WEAVER 2.0.

Process WEAVER is built around a centralized server by which workflow enactment is performed. Tool communication is realized by a broadcast message server (BMS, see chapter 2) through selective broadcasting of events to interested subscribers. Each workflow instance is executed in a separate process by a dedicated workflow script interpreter. However, each user logged in the system starts a local BMS for enabling tool communication within a user session. Users interact with the system through an agenda which uses the local BMS to synchronously or asynchronously invoke applications. Some aspects of the architecture of Process WEAVER is depicted in Figure 4-4.

TriGS_{flow}

The implementation architecture of TriGS_{flow} [Kappel et al., 1995] represents one of the most advanced approaches which use an ADBS as a workflow enactment and database server. It is built around the commercial OODB GemStone to which active functionality extensions have been made. The choice of a centralized ADBS however, hampers the scalability of the system as the server can quickly become a bottleneck when many clients concurrently execute multiple work-

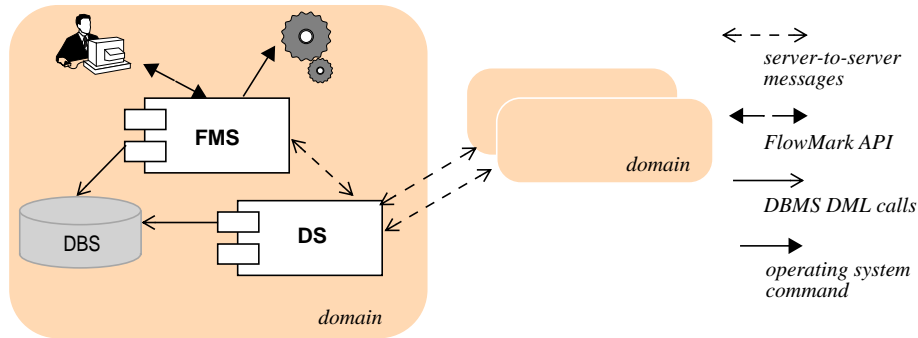


Figure 4-5: The implementation architecture of FlowMark-based workflow systems.

flows. While the extensibility of the system is well-served, the use of DBMS internal applications for the execution of workflow activities causes further load on the server. Clearly large scale distributed workflows cannot be supported by this system organization.

4.3.2 Replicated Servers

In this particular architectural variation, the workflow system comprises multiple identical servers. Each server has a workflow enactment and a workflow database subsystem, which contains complete information on the workflow types being executed. The main problem in this kind of is the efficient replication of workflow type information and the handling of the extensive network traffic that occurs during remote workflow execution. We consider two systems which follow this approach.

COSA

The workflow systems based on the WFMS COSA [COSA, 1998] of Software Ley have a replicated server architecture. They are built around *COSA Servers (CS)* which are groups of operating system processes running on a server host. Each CS is assigned a database server which can be located on any host of the system. Users/clients are assigned to a fixed CS and every server in the system has to know this assignment, i.e., the physical location of users/clients. Workflow models for running workflows are replicated on all servers. Due to the fixed allocation of users to servers, during distributed workflow execution, the complete instance data are migrated to a user's server—and deleted from the old server.

FlowMark

The FlowMark WFMS [Leyman & Altenhuber, 1994, Leymann & Roller, 1994] was originally developed as a fully centralized system but evolved to a distributed system with multiple workflow enactment servers. A FlowMark-based workflow system is partitioned in various domains consisting of multiple hosts. In each of these domains, a *Database Server (DBS)* is used to store workflow execution data, a *FlowMark Server (FMS)* process is responsible for workflow enactment and a *Delivery Server (DS)* is responsible for the communication among FMS. The FMS and DS are statically allocated to a DBS (and thus to the domain). The workflow execution operations are performed within database transactions. The architecture of FlowMark-based workflow systems is depicted in Figure 4-5. The distribution of workflow execution in FlowMark represents a hybrid approach. Distributed workflow execution is possible by associating a workflow type with a “home” FMS but allowing the execution of subworkflow types at different FMS which communicate through their respective DS. The distribution of workflow execution must thus de-

terminated by the workflow specification, a fact that renders the implementation architecture visible at the specification level. Furthermore, the operation of a domain and consequently the execution of allocated workflow types is dependent on the operation of the domain DBS.

The functionality of the FlowMark WFMS cannot be extended. The workflow system developer can only affect workflow execution through application programs accessing the provided API. Furthermore only specific platforms and DBMS are supported as provided by the manufacturer. Another commercial system which has a distributed multiple server architecture similar to that of FlowMark is CSE/Workflow [CSE, 1996].

4.3.3 Localized Servers

Many WFMS propose an operational architecture in which multiple servers exist, which are however localized close to where activity execution takes place. Systems belonging to this category include METEOR₂, Mentor, Mobile, WIDE, as well as our own approach as described in the chapter 9.

METEOR₂

In METEOR₂ [Miller et al., 1996, Miller et al., 1998] various system organizations have been developed with particular emphasis on scalability and large-scale fully distributed workflow execution. The system basically distinguishes between a build-time environment (the *designer*) and the workflow enactment environment. The build-time environment is separated from the execution environment by the use of an intermediate language in which task dependencies and invocation details are described. A complete workflow model is effectively mapped to a task dependency graph in which the workflow system components are aware of other components executing successor tasks and inform them upon the completion of preceding tasks. The scheduling code has to be compiled into the task managers during system initialization. This representation is subsequently used to generate an executable system which effectively consists of components called *task managers* implemented as CORBA objects. In METEOR₂ workflow systems a centralized monitoring server is informed about the execution of every workflow step. The monitoring service provides the basis for system recovery so that it must be always available in an operating workflow system. The architecture of METEOR₂ is depicted in Figure 4-6.

Task managers are automatically generated from the intermediate language and contain information necessary for task scheduling and task data transfers. They call code which implements application specific tasks, such as, wrapped legacy code, and application scripts. Each task manager provides task activation, task invocation (function call or process invocation) and output handling, as well as error handling and recovery. The main program for a task manager sequences

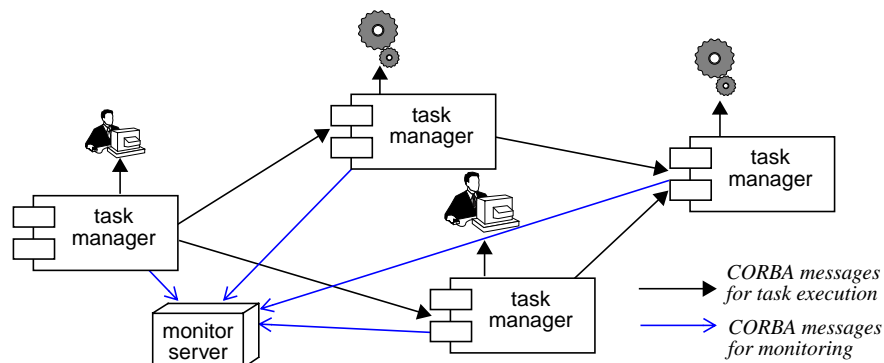


Figure 4-6: The implementation architecture of METEOR₂-based workflow systems.

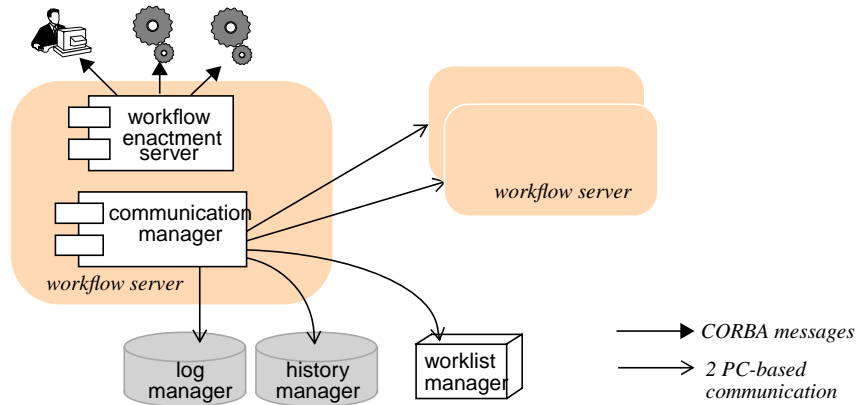


Figure 4-7: The implementation architecture of Mentor-based workflow systems.

through the initialize, execute and output methods of a task manager class. For each supported task type (automated or manual) there is different derived task manager class. Transactional automated tasks have the externally observable states initial, execute, abort, and commit while non-transactional automated tasks have the externally observable states initial, execute, fail, and done.

Mentor

In Mentor [Wodtke, 1997], actors are represented through elements of the state/activity charts used for workflow specification (see chapter 3). The basic idea underlying the implementation architecture is the partitioning and distribution of the state/activity charts in such a way as to preserve the semantics of centralized execution. Each step is executed by a workflow enactment server to which the actors responsible for the activity execution have been allocated. actors can be automated or interactive external applications. The behavior of interactive external applications in Mentor can be expressed by a generic state chart as depicted in Figure 4-8. The transition from init to ready takes place when input data have been read. The state active is reached when the user of the application interacts. This interaction can be interrupted in which case the state suspended is reached. When the interaction is completed the state OK is entered. Finally after output data has been written the state done is reached. From any of the states a failed state can be reached in which case the application has to be re-initialized. Variations of this standard application behavior can be described by removing some state; for example applications which do not produce data can be described by a state chart in which the done state is not present. When a transition fires in the workflow engine, the action part of this transition is executed, which may start an activity. The call is then forwarded through an activity stub, which represents the interface of the external system. The associated input and output parameters of the call to the external application are defined

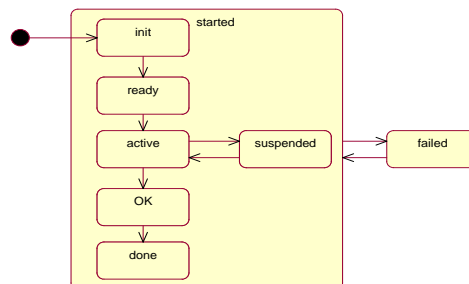


Figure 4-8: State diagram describing the behavior of invoked applications in MENTOR.

in this interface. External applications are considered as CORBA objects and consequently the communication with the external application is implemented through an ORB. As an advantage of this approach, the Dynamic Invocation Interface of the ORB can be used for late binding of application objects. However, this assumes that legacy applications have been wrapped with an IDL interface.

During workflow execution, the complete instance migrates from server to server exactly when the partitioning point of a state/activity chart has been reached. All internal communications are protected by a two-phase commit protocol [Bernstein et al., 1987]. The architecture of Mentor is depicted in Figure 4-7.

Mobile

In Mobile [Heinl & Schuster, 1996] a scalable multi-server architecture has been chosen which directly supports the proposed aspect-oriented workflow modeling approach [Jablonski & Busler, 1996]. Every aspect of a workflow (e.g. control flow and data flow) uses a different passive server object. These servers are called by a coordinating *operation execution server* or *Mobile kernel* which implements workflow operations. A workflow server consists of a kernel and various aspect server objects. Remote server objects are represented by proxy objects. Remote communication takes place by an RPC-style mechanism: OSF DCE and CORBA-based implementations have been proposed. The implementation architecture of Mobile is depicted in Figure 4-9

The scalability issue is addressed by workflow server replication, when the load of a particular server becomes excessive. Immutable workflow model data are replicated among each server. Every workflow type has a fixed workflow server assigned and workflow data are partitioned accordingly. Thus server replication may require the reassignment of certain workflow types. A workflow server and a workflow database is assigned to an organizational unit

WIDE

The WIDE project [Ceri et al., 1997] also proposes a distributed implementation architecture. Distribution is achieved by the use of a CORBA-based middleware infrastructure in which workflow execution engines are CORBA objects and database servers are encapsulated by an IDL-to-SQL mapping *Basic Access Layer (BAL)*. Each workflow engine in WIDE persistently stores workflow data in a relational DBS by using the BAL. WIDE-based workflow systems are divided in domains to which a workflow engine is assigned. A hierarchy of domains can be defined following the organizational structure of the workflow. The hierarchy provides various logical work-

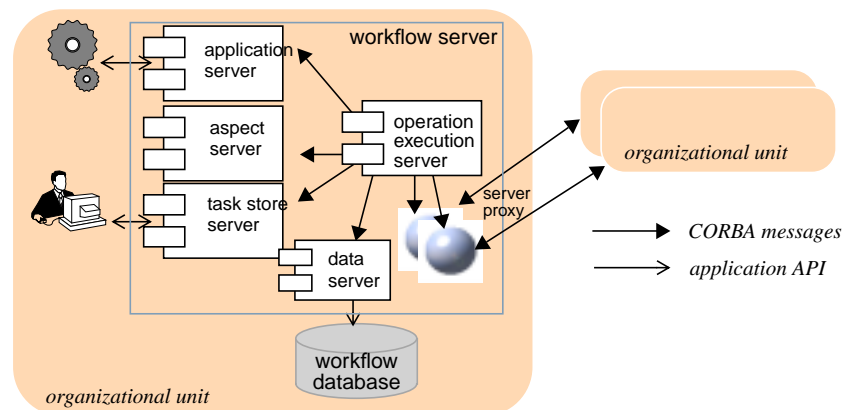


Figure 4-9: The implementation architecture of Mobile-based workflow systems.

flow execution layers and implements nested transactional workflow execution. No workflow data migration takes place and data are accessed remotely via the BAL.

WIDE uses rules for exception handling; rule management is performed by an *Active Rule Management* component. Events are detected by specialized detectors which then store them in an event database. A scheduling process retrieves the events from the database using the BAL and matches the events to rules which are subsequently executed by a rule interpreter component.

Workflow activities are executed by agents (see also section 3.5). Task execution is initiated by the workflow engine, which determines when a certain task must be executed, and controlled by the agent. Task assignment is based on agent attributes that are evaluated in task constraints. Two models for task assignment are distinguished:

- in the PULL model human users can actively select and execute a task from a shared pending task queue; and
- in the PUSH model task are forwarded by the workflow engine to the agents.

External applications are called by the workflow engine based on attributes of data elements. A document file for example, has as its attributes the operations `edit` and `print` which are strings containing program calls at the operating system level. The implementation architecture of WIDE is depicted in Figure 4-10.

4.3.4 Server-less Workflow Systems

The underlying idea in workflow systems without server subsystems is to migrate the server functionality to every participating client in the workflow system. Workflow instances migrate from one client to another during workflow execution. The state of the workflow execution is consequently distributed over the workflow system. A distributed synchronization mechanism is required to determine the actors which execute workflow activities.

Exotica/FMQM

The Exotica/FMQM system [Alonso et al., 1995] has a distributed server-less workflow system architecture. A reliable communication system based on persistent message queues is used to connect *processing nodes*, which contain those parts of the workflow specification graph that contain activities performed by the users local to the processing nodes. Graph partitioning takes place during workflow specification. After activity execution is completed, messages are sent to all sub-

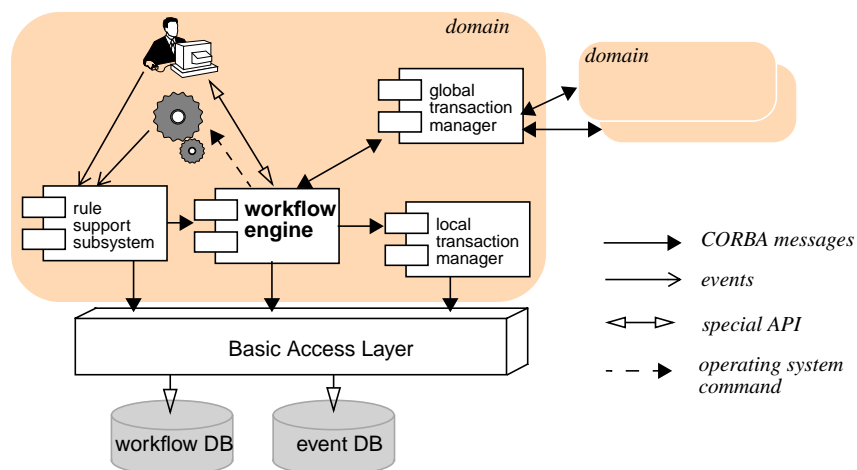


Figure 4-10: The implementation architecture of WIDE-based workflow systems.

sequent processing nodes which are defined in the activity graph. These may choose to retrieve the results of the activity from the output queue of the previous node. Node synchronization is thus achieved by means of the transaction system provided by the queuing system. The architecture of Exotica/FMQM is depicted in Figure 4-11.

The main problem with this approach is that of excessive network traffic. This occurs since data produced by an activity has to be sent to all nodes in the system as it is not known on which eligible node the activities will actually execute before node synchronization has taken place.

Agent-Based Workflow Execution

Agent-based workflow execution approaches have been influenced by artificial intelligence software agent research and emphasize dynamic change of running workflows as well as intentionality. “Agenthood” is an overloaded term in workflow management; we use it in the sense of mobile, goal-oriented software entities. Agents are characterized by being able to suspend their execution at some point, transport themselves to another connected machine, and resume execution on the new machine.

The main advantages claimed by proponents of agent-based workflow execution are the complete decentralization of control flow, and the flexible evolution of workflows. It is questionable however, if due to the self-modifiability of agents any control at all remains at the end of the day! Furthermore, the enforcement of security policies is rather problematic as it has to be ensured for every processing station. The use of a programming language with appropriate security features (e.g., Java) may alleviate or reduce this problem to some extent. Finally, an issue which has to be considered in agent-based workflow systems, concerns the high network load resulting during agent migration. Examples of agent-based approaches to workflow execution include INCAs [Barbara et al., 1996], the proposal of [Borghoff et al., 1997], and WorkWeb [Tarumi et al., 1997].

INCAs introduces the concept of *information carriers* which are essentially mobile, self-modifying activity representation entities [Barbara et al., 1996]. They consist of private data defining a global workflow context shared among the constituent steps of the activity, historical data on the preceding INCAs execution, and rules defining the control and data flow between the steps of the activity. The execution of these activities on the *processing stations* to which INCAs arrive may result in new rules being added to the INCAs or existing rules being modified. A reliable communication infrastructure is used so that processing stations do not have to consider security aspects *during the transportation* of INCAs.

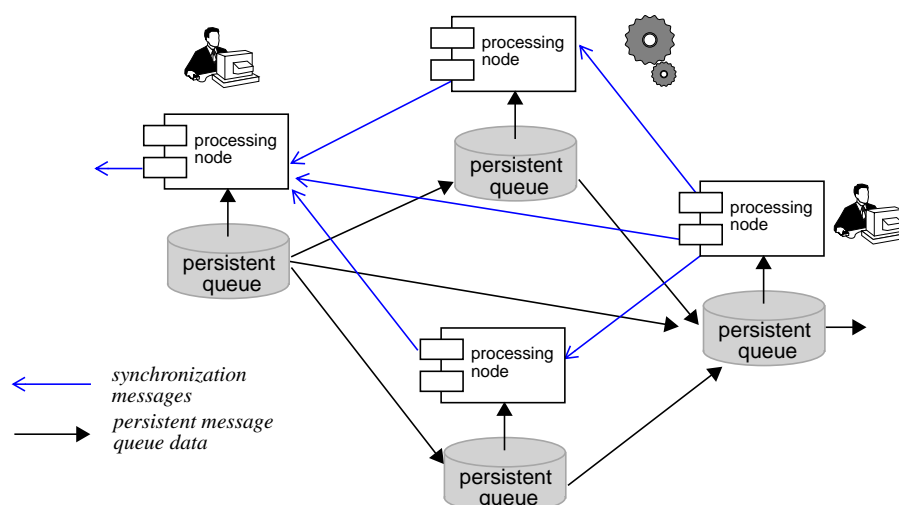


Figure 4-11: The implementation architecture of Exotica/FMQM-based workflow systems.

In [Borghoff et al., 1997] *reflective agents* are used which exhibit reactive behavior to execute workflows and *deliberative behavior* in order to modify their own plans. Thus a reflective agent has representations of its current state, its abilities (i.e., the actions it can perform), its past execution history and the *goals* it is pursuing. The explicit representation of goals belongs to the intentional perspective of the workflows. An agent rule specifies an activity whose state is encoded in attributes accessed through the agent's state. Pre-conditions and post-conditions of the activity are represented as simple non-computational resources.

4.3.5 Evaluation

In the previous section we considered four different classes of WFMS implementation architectures: systems based on a centralized server, systems with identical replicated servers, systems which locally adapted servers, and server-less systems with fully distributed flow of control. While none of these architectural variations can be considered superior each one of them is better suited for particular classes of workflow application systems and for the non-functional requirements which the application poses.

We conclude this section by characterizing the various architectures with respect to various *architectural quality attributes* [Bass et al., 1998] which are particularly relevant for WFMS. These properties are architectural in nature in the sense that they are affected by the architectural attributes of the system. This does not mean however, that they are only dependent on the architecture of the system.

- *Scalability.* This property refers to the limitations of the size of the workflow system with respect to the number of concurrently executing workflows and with respect to the size of the organization in which the workflow system operates (i.e., the number of application actors and distributed sites).
- *Availability.* This property refers to the guarantees that the WFMS makes with respect to failure resilience and fault tolerance. It is related to the provided flexibility of assignment of workflow tasks to particular servers as well as with the redundant components available in the system.
- *Modifiability.* This property refers to the possibility to modify and extend the basic WFMS functionality as required by the concrete workflow application system. Different classes of workflow management applications have different requirements on the underlying execution infrastructure. The WFMS should be able to accommodate desired extensions. The modifiability of a system is directly dependent on the locality of change. This requirement is also related to integrability (see below).
- *Integrability.* This property refers to the support for workflow execution over heterogeneous software systems through the provision of appropriate integration mechanisms. These include various kinds of business applications, middleware services, as well as the interoperability with other WFMS. Integrability is a function of the integration mechanisms provided by the WFMS and depends on the ways in which the functionality of actors is accessible to the workflow system (e.g., consider a simple call to an interactive application compared to white-box integration).

Table 4-3 presents an overview of the architectural quality attributes of the described WFMS architectures. Low means that the architecture provides little or no support for the quality attribute, while high means that the architecture effectively supports the quality attribute.

It is obvious from the table, that systems based on a centralized server have low scalability. Furthermore, we note that only few systems provide mechanisms to extend the basic WFMS functionality the notable exceptions being TriGS_{flow} which allows extensions to be built as OODB applications and Mobile which provides an extensible WFMS architecture.

Table 4-3: Qualitative evaluation of the architectural properties of various WFMS.

<i>System</i>	<i>Scalability</i>	<i>Availability</i>	<i>Modifiability</i>	<i>Integrability</i>
ProcessWEAVER	low	low	low	low
TriGS _{flow}	low	low	medium	low
COSA	low	low	low	medium
FlowMark	medium	low	low	medium
METEOR ₂	high	medium	low	medium
Mentor	medium	low	low	medium
Mobile	medium	low	high	medium/high
WIDE	medium	medium	low	medium
Exotica/FMQM	high	low	low	medium
INCAs	high	low	low	?

4.4 A Classification Scheme for Workflow System Actors

In this section we propose a systematic actor analysis and classification scheme. The scheme forms the basis of the systematic approach towards workflow system composition used in the RE-WORK metamodel by identifying common characteristics of types of actors. The use of this classification scheme by the workflow system composer (see section 4.1) allows the subsequent integration of actors in a workflow system by the identification and selection of predefined integration components.

4.4.1 Purpose and Overview

As already described in the previous chapters, application metamodels in most existing workflow specification environments abstract from the underlying application technology. The advantage of this approach is the separation of concerns between specification and implementation. On the other hand, the developer of a workflow application system receives little support for the workflow system composition at a conceptual level. By providing a framework for the systematic characterization of workflow application systems, the complex integration of various applications but also the extension of the kernel WFMS with additional functional components can be reduced to more general terms for which predefined integration patterns can be provided. Our classification approach can most closely be compared with that of Mobile [Jablonski & Bussler, 1996]; however, in Mobile, only workflow applications are systematically characterized, while WFMS components are not explicitly defined as part of a workflow system architecture. Furthermore, while the emphasis in Mobile lies on the specification—with the ultimate goal of defining appropriate wrappers—in our approach we define constraints on the correctness of the architecture and the roles of actors in the system operation.

The classification scheme is derived from the multi-faceted classification scheme proposed by [Prieto-Díaz & Freeman, 1987] for the classification of reusable software artifacts. Our classification scheme proposes an actor description format based on a standard *vocabulary* of terms and imposes a citation order for the facets called *actor attributes* or simply attributes. Unlike the pure faceted classification, leafs of the classification trees may be either be *simple terms* or *expressions*. Simple terms are chosen from the standard vocabulary of the domain, while expressions describe a dependency to other elements of the workflow system architecture —elements which are not necessarily components. Such elements may be defined outside the scope of the classification scheme, (e.g., in the workflow model).

Similar classification approaches have been used in various computer science domains. [Gepert, 1994] proposes a faceted analysis scheme for DBMS components. DBMS *features* describe aspects of reusable artifacts consisting of a functional domain description based on well-known terms in the database domain. The classification scheme is applied to transaction management subsystem specification. In the IPSEN project stored development software documents are classified by a feature-oriented scheme [Börstler, 1992]. The semantics of a feature are given through successive refinement down to the level of well-known notions corresponding to terms of other classification schemes. View-based, decompositional, and specialization refinement are supported.

The facets identified for a particular classification scheme depend on the particular purpose of the classification. Furthermore, they depend on the required support for (automated) reuse. In our work we can identify four main purposes for the classification:

- development of a conceptual basis for the workflow system metamodel proposed in the next chapters,
- determination of the properties that the component templates used for actor representation must have,
- analysis of the integration issues facing the composer of a workflow system, and
- architecture-centric based development and reuse [Boehm & Sherlis, 1992].

Through the use of the classification scheme described in this section, workflow system actors can be characterized with the help of standardized attribute terms and expressions pertaining to their participation in the workflow system (e.g., the functionality they provide), and the required integration mechanisms (the way they provide it, their interfaces, etc.). Contrasting our approach to the description of software artifacts, in the form e.g., of software code, we note that the central aspect, i.e., the domain of the attributes, is the interaction description. The requirements are similar, in this sense, with those underlying the development of *gluons* for real-time financial applications as advocated by [Pintado, 1995]. The classification scheme provides a first impression to the workflow system composer on the interaction patterns between the various actors focusing on the classification axes of control and data interaction. Questions such as the following, may be answered:

- Is the actor required for workflow execution?
- How many instances of the actor may coexist in a workflow system?
- What kind of connectors/interaction mechanisms does the actor support?
- What properties of service execution by the actor can be assumed?

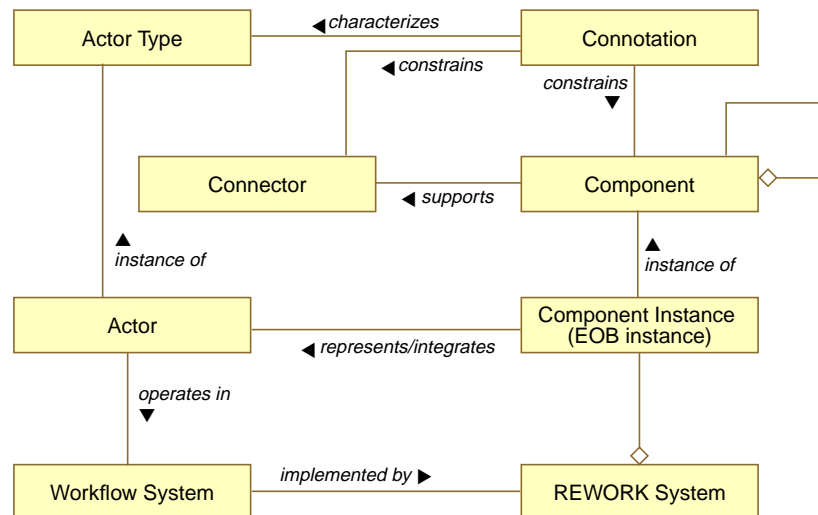


Figure 4-12: A class diagram [Booch et al., 1999] depicting the relationships between actors and workflow system components as defined by the REWORK metamodel.

The defined values of the attributes of an actor compose the actor connotation. Actors with the same connotations belong to a particular actor type. Thus, a connotation characterizes the actor's role in the workflow system in a standard terminology. It externalizes the actor properties with respect to its participation in a workflow system.

Compared to an architecture definition language, such as for example Rapide [Luckham et al., 1995], the actor classification scheme provided here can be considered more of an analysis than a specification mechanism. In that sense, the actor connotations are constructs which constrain the implementation of workflow system components and connectors. The components and connectors in the architectural framework defined in the next chapter implement actors and their interactions. Generally speaking, actors are mapped to multiple components and connectors in the architecture of the composed workflow system (see also Figure 4-12).

During the analysis there are a variety of participants who must provide information to build the classification scheme. In our case the following sources have been considered:

- *End users.* The people that use systems in the domain. These participants know how the systems they use operate, understand where the systems fit in a larger flow of control, and what capabilities are missing from current systems.
- *Domain experts.* The personnel that provide information about systems in the domain. They include vendors of WFMS, members of the WFMS research community and developers of applications which are integrated in workflow systems.
- *Workflow applications.* Analysis of existing workflow applications as for example, provided in the case studies. Furthermore, analysis of the requirements for the development of new workflow management applications.
- *Domain-related publications.* These include published descriptions of research prototypes and products alike.

4.4.2 Attributes of Workflow System Actors

In this section, we describe the attributes provided by the REWORK metamodel which may be used for characterizing the various actor types. The attributes belong to two categories: *participa-*

tion attributes characterize an actor from the perspective of its participation in the workflow system; *implementation attributes* characterize the actor from the perspective of the support required by representing components. Each attribute vocabulary contains a list of terms which can either be simple terms or expressions. We note, that not every term combination is meaningful. We also note, that the assignment of a particular term value to an attribute has implications on certain aspects of the overall system architecture.

Participation Attributes

Participation attributes characterize an actor's role in a particular workflow system. Thus, the same actor may be assigned different terms according to the workflow system in which it participates. The participation attributes define architectural constraints on the workflow system.

Optionality. This attribute refers to the requirements concerning the existence of the actor. The implications of the attribute concern the existential properties of components representing the actor in the workflow system. The following terms/expressions are defined:

- Required: the term denotes that the actor must always be present in the system as it provides functionality required for every workflow executed in the system. For example, a timestamp creation component is always required for workflow execution.
 - The completeness of the workflow system can be examined over the set of components implementing the required actors.
 - It must be examined whether such actors belong to the workflow system infrastructure, i.e., are required across a range of workflow application systems.
 - The components integrating the actor must be present in the workflow system.
- Optional (*optional descriptors*): the expression denotes that the actor provides functionality which is sometimes needed for workflow execution. For example a workflow logging component is only required if logging functionality must be provided.
 - If a specific functionality must be provided, the components implementing/representing the actor must be present in the workflow system.

Multiplicity. The attribute refers to the number of instances of the actor which may be concurrently active in a workflow system. The implications concern the required task assignment protocols. The following terms are distinguished:

- Singleton: the term denotes that exactly one actor of this type can be active at any time in the workflow system.
 - Only one component representing the actor can be created at system instantiation, and further component creation must be disabled.
- Multiple (*comparison_operator expression*): the expression denotes that multiple instances of the actor may exist concurrently. The maximum number of instances may be optionally defined. For example, a worklist and the respective interface is required for each staff member participating in the workflow.
 - The creation of components representing these actors must be controlled.
 - Appropriate control and administration mechanisms must exist in the workflow system.

Dependency. The attribute refers to the assumptions made by the actor for its operation with respect to the existence of other actors. The following terms/expressions are distinguished:

- None: the term denotes that no assumptions are made by the actor regarding the existence of other actors. The actor operates autonomously.
- Dependent (*reference list*): the expression lists the actors which must be available for the correct operation of the dependent entity.
 - The existence of the listed actors must be ensured.

Function. In the analysis, this attribute is included to provide a general characterization of the component's role in the workflow system. The various functions/services exported by the actors will be used directly or may be composed to higher-level services in the workflow system. Often, multiple services are provided by one actor and attributes are parameterized with descriptive details of the service. The term vocabulary in this attribute is open-ended but includes the following (from more general to more workflow management-specific):

- Display: services related to visual operations such as display, hide, animate.
- User interaction: services for manipulation on-screen elements, e.g., select and input.
- Persistence: services related to the storage of workflow and administrative data.
- Transformation: services related to transformation between data formats.
- Transactions: services related to adding transactional properties to non-transactional service executions, as for example provided by TP monitors.
- Query: services related to the retrieval of stored data.
- Notification: services for sending notification messages to interested recipients.
- Time: timestamp creation, alarm, etc.
- Enactment: services related to the task execution and control flow enforcement.
- Log: services related to logging workflow execution, analysis of logs, etc.
- Worklist: services related to the management of tasks assigned to humans such as worklist actualization and synchronization.
- workflow application specific functionality.
 - Appropriate connectors must be defined to access these functions.

Implementation Attributes

Implementation attributes constrain the components of the workflow system which represent/implement the actor.

Automation. The attribute refers to the automation degree of the actor. The following terms are defined:

- Manual: denotes that the actor is a human being.
 - A component supporting person/machine interaction must be provided. This component has to provide the user information about tasks the user must execute and transfer the results back to the workflow system.

- **Interactive:** denotes that the actor is an interactive application. The completion of execution of services by the actor takes place with the participation of human beings. The interaction can take place solely at application start or during the entire application execution.
 - Connectors and components must exist to catch asynchronous user interaction.
 - The users interacting with the application must be represented in the system.
- **Submit-and-forget:** denotes that the actor operates without human intervention. It may be either an invoked application or a workflow system component. Note that the term is not equivalent to a batch connector of the application (see below).

Context. The attribute refers to the capability of an actor to maintain (retrievable) service execution context between successive executions. The mechanism used to store context information is not important. The following terms are distinguished:

- **No:** denotes that the actor does not maintain context information between successive activity executions. This usually implies, that the actor depends on its existence on an executing operating system process.
 - If context information is required between successive service executions it must be kept by some component in the workflow system.
- **Yes:** denotes the actor can keep context information between successive service executions. It also provides functionality to retrieve this context.
 - The component representing the actor must be able to access the state information.

Server type. The attribute refers to the properties of an actor with respect to providing its functionality to other actors. The following terms are distinguished:

- **Invoked:** denotes that the actor is not a server process but must be explicitly invoked to provide some functionality. The actor process runs only for the duration of the particular request servicing.
 - The appropriate application invocation mechanism must be implemented in the representing component.
- **Shared:** denotes that multiple services can be requested concurrently from the actor without waiting the completion of previous services executed by the actor. The actor can be shared by multiple clients.
 - Transparent actor invocation must be supported by the representing component.
 - If concurrent service execution is required in the workflow system, the component representing the actor must support non-blocking service execution (see chapter 6).
- **Blocking:** denotes that only one service at-a-time can be requested from an actor.
 - The component representing the actor must allow only blocking service execution. Further requests have to be rejected or must be diverted to another actor.

Execution guarantee. The attribute refers to the guaranteed properties of service execution by an actor.

- **None:** denotes that the actor does not provide any guarantees for service execution.
- **Yes (optional descriptors):** the expression denotes the guaranteed execution properties of services provided by the actor, e.g., ACID.

- The component representing the actor must provide appropriate execution guarantees.

Connector types. The attribute refers to the mechanisms(s) of interaction that the actor supports (the ingoing interface) and the mechanisms of interaction it uses (the outgoing interface). The different kinds of interactions imply for the system architect that a different kind of communication (and eventually wrapping) technique must be used. The appropriate connectors must be supported by representing components.

The vocabulary is open-ended but includes the various standard connector types (e.g., [Bass et al., 1998]). Each attribute may have optional descriptors characterizing the access interface. The following is a list of standard expressions:

- Batch (*optional descriptors*): batch file-input. The descriptors may refer to the batch file format, and other batch submission properties.
 - A component that can create and read batch files with the given format and supports the batch job submission protocol must be defined.
- Local procedure call (*optional descriptors*): one thread of control in a single name space. The descriptor may refer to the libraries used, the programming language, etc.
 - A component that is linked with the libraries must be defined.
- Remote procedure call (*optional descriptors*): one thread of control in separate name spaces. This indicated the support for a mechanism that allows clients to execute code that resides in a server, regardless of whether the server is executing locally or remotely. Example descriptors include DCE RPC and Sun RPC.
 - An RPC calling system (client) component must be provided.
- Remote method invocation (*optional descriptors*): method invocation in objects residing in a separate name space. Example descriptors include CORBA-compliant systems (e.g., Orbix), or Java RMI.
 - A remote method invocation client component and appropriate middleware infrastructure must be provided.
- Message protocol (*optional descriptors*): explicit discreet handoff of data between independent processes. Example descriptors include POP and HTTP.
 - A component capable of sending and receiving messages abiding to the protocol must be defined.
- Implicit triggering (*optional descriptors*): the computation is invoked by an event occurrence. The descriptors refer to the event format, for example, X-window events or Apple events [Apple Computer, 1991].
 - A component that registers with the event mechanism used by the actor and asks to receive or capture certain events must be defined.
- Operating system call (*optional descriptors*): program (new process) invocation with parameters. The descriptors refer to the operating system mechanisms used.
 - A component that can start an operating system process for the platform of the actor must be provided.

- Script (*optional descriptors*): actor services are invoked by script commands. The descriptors refer to the script language understood by the actor.
 - A component that can submit script commands and read the results of the actor computation (e.g., from a file) must be provided.
- Shared data (*optional descriptors*): interaction over a blackboard, a shared file, or database. The descriptors refers to a file name, data format, database access protocol, etc.
 - A component that knows the shared data format and can read and write in the shared storage system must be provided.
- Dynamic query (*optional descriptors*): the actor provides a rich query and update mechanism via some language. The descriptor refers to the language query can be a standard such as SQL or a vendor proprietary language. The ODBC and JDBC database connectivity standards are prominent examples.
 - A component which can generate query language statements, pass the to the system and receive and parse results must be provided.

Inevitably, we dedicated a lot of time considering the completeness and orthogonality of the attributes by which we structured the classification of actors. It is however possible, that additional attributes may be required and even within attributes which seem complete, i.e., all except those concerning interface and service, additional terms and expressions may be required. Such modifications and additions are not excluded by the classification scheme. However, the implications and effects they have on the other aspects of the REWORK metamodel will have to be considered and the new attributes must be represented in the build-time repository (see chapter 8). In order to demonstrate attribute-based classification, we classify actors from the examples of section 4.2 as well as other workflow systems.

4.4.3 Examples of Actor Connotations

In this section, we define example connotations both for external applications and WFMS-internal actors. We note, that during the classification of actors some attributes can be left unspecified. This means that the workflow system composer has to manually ensure that the correct type of components is chosen and that all necessary components are defined in the system architecture.

Mail Handling Workflow System

We consider the HAI application from the second example system (see section 4.2.2). Its connotation in the workflow system is described in the following table:

Connotation: HAI (Mail Handling Workflow System)

Participation	Optionality	required
	Multiplicity	singleton
	Dependency	none
	Function	invoice control
Implementation	Automation	interactive
	Context	yes
	Server type	shared
	Execution guarantee	ACID
	Connector types	in, out: remote method invocation (Visibroker IDL)

While connotations can be used to characterize actors participating in a specific workflow system, they can also be used to identify characteristics of actors which are considered to be part of the infrastructure of a workflow system and consequently are not explicitly described in other approaches. This is exemplified in the next sections.

Worklists

Often workflow tasks are performed by human beings. Such tasks may be purely manual and involve human decision making, informal communication, and interactive tasks performed with the help of some application. In any case, the results of the task execution must be communicated to the workflow system. Humans interact with the workflow system through *agendas* which are usually graphical user interfaces providing them with access to assigned tasks. The required WFMS functionality consists of (e.g., based on [WfMC, 1994]):

- *user interaction*: consists of displaying pending tasks and their parameters and accepting user selections (eventually leading to the execution of some application); and
- *worklist management*: includes storage of tasks assigned to a user, deletion of completed tasks, modification of task priorities, etc.

In the WfMC reference model, four alternative scenarios for worklist handling are proposed which differ with respect to the centralization of worklist management. They define the coupling between component which is responsible for administering the worklist and the workflow enactment engine [WfMC, 1994]. These scenarios are the following:

- *host-based model*. The worklist client is on the same site as the workflow engine and communicates through a local interface. The worklist is managed by the workflow engine. It corresponds to the connector type remote procedure call.
- *shared file store model*. The worklist handler is a client of the workflow engine storage system. It corresponds to the connector type shared data.
- *electronic mail model*. The worklist handler is an autonomous application which communicates with the workflow engine through message protocol.
- *“procedure call or message passing model”*. The worklist can either be administered by the workflow engine or the worklist handler. It corresponds to the connector type remote procedure call.

We can characterize, for example, Process WEAVER agendas [Fernstrøm, 1993] with the following connotation:

Connotation: Agenda (Process Weaver)

Participation	Optionality	required
	Multiplicity	multiple (\leq number_of (local_BMS))
	Dependency	local_BMS, user
	Function	user interaction, display, worklist management, notification
Implementation	Automation	interactive
	Context	yes
	Server type	shared
	Execution guarantee	none
	Connector types	in, out: implicit triggering (BMS messages)

Wide Basic Access Layer

In Wide [Ceri et al., 1997] a relational database system is used for the storage of workflow data. An encapsulation layer, or *Basic Access Layer (BAL)*, provides a CORBA-compliant interface to other components of the workflow system and to the workflow engine. It does the mapping from the object-oriented data specifications to the relational database manipulation operations. The BAL uses a call-level interface to access the relational database system.

Connotation: BAL (WIDE)

Participation	Optionality	required
	Multiplicity	single
	Dependency	Oracle server (WIDE)
	Function	database access, interface transformation
Implementation	Automation	submit-and-forget
	Context	no
	Server type	shared
	Execution guarantee	ACID
	Connector types	in: remote method invocation (Orbix IDL) out: remote procedure call (Oracle CLI)

4.5 Summary

The conceptual classification of actors provided by the REWORK metamodel provides mechanisms for the analysis of their role and functionality in a particular workflow system. Actors are described by standardized terminology organized along different attributes. The resulting description of the actor provides decision support for the identification of standardized component templates and connectors required for its representation/integration in the workflow system. The composition of concrete workflow systems is subsequently based on the customizing and instantiation of the provided component templates.

5 Events in Workflow Systems

In this and the next chapter we describe the REWORK architectural metamodel. The metamodel borrows its principal style elements from ADBS and implicit invocation architectures; it provides abstractions for the description of workflow system actors by components and connectors. The metamodel provides a conceptual framework and the appropriate tools for the specification of both the structure of a particular workflow system and the behavior and interactions of participating actors. The resulting specification of the workflow system—called a REWORK specification—is rendered executable by its instantiating on top of an appropriate event engine provided by the REWORK environment. The resulting system is called a REWORK system.

Asynchronous event-based communication and interaction mechanisms for components are provided. We motivate the event-based component integration in REWORK systems by describing its advantages compared to other interaction mechanisms, and demonstrating how events can be advantageously used in workflow systems. We subsequently consider in detail the component connectors provided by the REWORK metamodel: services and event types.

We initially provide some informal definitions as a basis for the subsequent discussions.

Definition 5-1: (REWORK metamodel)

The *REWORK metamodel* comprises a set of conceptual tools for the specification of a workflow system architecture.

The application of the conceptual tools provided by the REWORK metamodel results in a REWORK specification. This specification consists of a collection of artifacts which are stored in a build-time repository.

Definition 5-2: (REWORK specification)

A *REWORK specification* is a specification of a workflow system created with the conceptual tools provided by the REWORK metamodel.

The instantiation of a REWORK specification is a process through which the run-time counterparts of specification artifacts are created. The run-time artifacts are objects stored in a distributed run-time repository. These run-time objects and their permissible interactions define the software architecture of a REWORK workflow system.

Definition 5-3: (REWORK system)

A *REWORK system* is an instantiated REWORK specification. It implements the workflow system specified by that REWORK specification.

This chapter is structured as follows: in section 5.1 we motivate the basic event-based style we have chosen for the REWORK metamodel. In section 5.2 we define services as the abstraction used to describe the functionality of reactive components representing actors. Services provide

the basic constructs for the description of functional aspects of a workflow system which can be leveraged to the high-level specification of workflows. In section 5.3 we introduce the notion of events as the basic connectors between workflow system components. In section 5.4 the basis for the formal foundations of workflow execution by components is set with the introduction of composite events. Section 5.6 concludes the chapter by formally defining composite event semantics and event histories over which the correctness of the actor behavior can be expressed.

5.1 Motivating an Event-Based Workflow System Metamodel

At the most abstract level, a workflow system is a distributed system consisting of a collection of *actors* and an *integration infrastructure* capable of providing coordination functionality and communication channels between the actors for the exchange of data. As we have assumed in the previous chapter, the heterogeneity of the actors which make up the workflow system is analyzed and mapped to properties of REWORK system components. Each actor is represented by the same class of a reactive composite component in a REWORK system: an *event occurrence broker* or *EOB*¹.

An EOB is informed of events occurring in the workflow system for which it has registered an interest, and defines reactions that transform them to interaction primitives understood by the actors, leading to workflow task execution. Furthermore, it returns the results of task execution to the workflow system in the form of events. The EOB which have registered an interest for an event type are called the event type's *subscribers*. Thus:

Definition 5-4: (Event occurrence broker)

An *event occurrence broker* is a reactive aggregate component described by a REWORK specification. The component executes in a REWORK system.

5.1.1 Characteristics of the REWORK Metamodel

In developing an appropriate component metamodel, we considered a set of requirements with respect to its expressiveness and flexibility. In this section we summarize these requirements. The REWORK metamodel must allow:

- the definition of the actors composing the system. In the REWORK metamodel these are represented by aggregate components called EOB;
- the definition of those properties of the actors which are relevant for their integration in the workflow system. These include access interfaces and execution properties and represented by the properties of EOB;

¹ In our earlier publications (e.g., [Tombros et al., 1997]), the simple term “broker” was used in place of EOB. The previous use of the term has its origin in the terminology used in [Geppert, 1994], where brokers represent subsystems of a DBMS providing services to other parts of the DBMS. Although the basic notion of brokers—in the particular sense mentioned above—being subsystem representations remains, in this thesis, we introduce the term “EOB” in order to denote the particular emphasis on event-based interaction and event-based invocation of component functionality. The similarity to the term “ORB” Object Request Broker [OMG, 1995] is not accidental. An ORB accepts and forwards object requests (i.e., messages) to server objects while an EOB forwards event occurrences to actors.

- the definition of and access to workflow-related functionality of actors; and finally
- the modification of a given workflow system architecture in response to evolving environmental requirements;

Especially concerning the last point, REWORK specifications have the following properties:

- they enable the reuse of individual component specifications and component configurations to construct new artifacts;
- they allow the evolution of the system as process and technological requirements change;
- they allow the definition of constraints imposed on the EOB composition, the invariants of the overall architecture, and the permissible EOB interactions.

A workflow system is a large complex software system. In order to keep complexity under control the REWORK metamodel provides the following support:

- it distinguishes between a *kernel system* and *extensions*, so that a workflow system can be built and extended incrementally;
- it dictates a modular system design according to locality principles which accommodate change and allow for a manageable balance between distributed and local functionality within components; and
- it defines variable and invariant properties of each REWORK specification which can be checked at defined points during the system life cycle.

These characteristics are manifested in the types and properties of components and connectors provided by the REWORK metamodel, in the provided component composition mechanisms, the operations that can be performed on components, and in the permissible component interactions in the resulting REWORK system.

5.1.2 Event-Based Integration

The most flexible and loose model of integration is *asynchronous interaction* based on *events*. The event-based approach to data, control, and process integration in REWORK systems is an extension to event-based architectures such as those described in chapter 2. Its novel features include the use of composite events and the distribution of the event infrastructure. The approach has the following advantages specific to its use in workflow management:

- It is a powerful paradigm allowing the expression of complex interaction patterns. Event-based interaction can be used to implement other kinds of interaction, for example, synchronous and asynchronous message passing, selective broadcasting, etc.
- It facilitates system evolution due to the loose coupling of components. Since events represent the sole interaction mechanism, the need for explicit inter-component references is avoided.
- It allows the description of different kinds of situations by appropriate event types (system internal, external and complex) in a uniform way. The description of component behavior can consequently be specified in a declarative way by ECA-rules triggered by those event types.

- Workflow monitoring can be implemented efficiently and without performance penalties [Schwidorski, 1996]. It is also straightforward to implement logging of workflow execution simply by persistently storing event occurrences.
- Formal workflow models can be discovered by mining event data. This may be useful, for example, when exception events occur too often, or for the specification of frequently recurring ad hoc workflows. For a detailed discussion of this issue in the context of software processes we refer to [Cook & Wolf, 1995]. Their work can be readily adapted to workflows.

There are however some disadvantages to event-based integration which can be summarized as follows:

- Events may lie at a relatively low level of abstraction. Thus, higher-level constructs are needed in order to render events useful at an architectural level. Despite the fact that events in the REWORK metamodel are not comparable to low-level events such as those in distributed debugging systems, they are not directly used for workflow modeling. Instead, we introduce the concept of services (see section 5.2) as a means of mapping events to actor execution.
- Events depict state changes and as such state is only modeled implicitly. Although this may sometimes be a disadvantage with respect to state-based workflow modeling (e.g., to express certain modeling constructs such as *implicit OR-split* [van der Aalst, 1998]), the definition of appropriate composite event types and behavior of participating actors can alleviate such problems to a large extent (see chapter 7).

In REWORK systems events are used for component integration and interaction, i.e., they are the connectors between EOB. Thus summarizing, events are used for the following purposes:

- signaling of the occurrence of workflow situations such as the completion of one or more processing steps, leading to notification of EOB; this leads to the invocation of functionality provided by the represented actors (see section 5.2);
- exchange of data between actors through event parameters (see section 5.3); and
- definition of control flow through composite events (see section 7.1).

A complete REWORK system assumes the existence of an *event engine (EVE)*. EVE is a distributed glue system which provides event-related functionality (transparent message transfer, event composition, and subscriber notification), as well as global state server functionality to EOB.

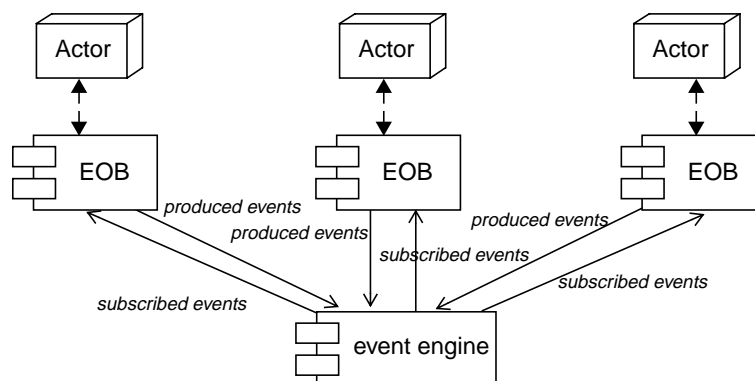


Figure 5-13: The high-level conceptual architecture of a REWORK system. Plain arrows are REWORK event connectors, and dashed arrows are actor-specific types of connectors.

From the perspective of the REWORK metamodel, EVE is an EOB which by default is defined in every REWORK specification. Similarly to other EOB, EVE represents a workflow actor, in this particular case the WFMS kernel. EVE is a subscriber to all defined event types and can define reactive behavior in response to the occurrence of workflow situations.

Invariant 1: A single event engine must exist in every REWORK system.

The use of EVE in REWORK specifications is described in chapter 6. The implementation of the event engine and its operation in a REWORK system is described in detail in chapter 9. The resulting conceptual organization of a REWORK system is depicted in Figure 5-13. Note that the actors are both external workflow application systems and components providing WFMS functionality. The strict distinction between workflow management infrastructure and workflow applications is relaxed allowing the seamless extension of WFMS.

The types of events available in the REWORK metamodel, their use, as well as their formal semantics are described in the following sections. REWORK components are described in the next chapter. Prior to describing REWORK events however, we consider the representation of workflow tasks through services.

5.2 Services and Workflows

The term “service” is used in many domains of information technology. *Services* in the REWORK metamodel express available system functionality. They are defined based on the function attributes of actors (see section 4.4). In a REWORK system a specific service is provided by one or more EOB—called the *server(s)* in the context of a service execution. It can be requested by EOB—called the *clients* in the context of that execution. Services define the functionality provided by actors in a given workflow system; service implementation may be, however, dependent on the EOB representing the actor. Services can be referenced in workflow specifications as atomic activity types declared in a workflow model. In other words, workflow specification makes use of the service abstraction provided by the REWORK metamodel in order to define the functionality for desired processing steps. In this way, the gap between workflow specifications and the defined system architecture is bridged.

A service is described by a request signature with a predefined set of parameters which are bound to values by the client EOB during service request. A set of possible replies is defined for a particular service, which express the possible outcomes of a service execution. These replies may have various parameters bound to values by the server EOB. In order to define services, we first need to define parameter declarations. Parameter declarations indicate the type of parameters used in services and events (see below).

The set T of defined parameter types available in the REWORK metamodel provides the types integer, float, char, boolean, octet-stream, date, and stream. These are defined as follows:

Definition 5-5: (Parameter types)

The set T of parameter types consists of the following elements:

- an integer data type represents the range $-2^{31} \dots 2^{31}-1$
- a float type represents IEEE single-precision floating point number (ANSI/IEEE Std 754-1985)
- a char type represents a single 8-bit character

- a boolean type represents the values TRUE or FALSE
- an octet-stream type represents an 8-bit quantity that is transmitted between workflow system sites without undergoing any transformation.
- a date type is a string of the form ddmmyyy-hhmmss (day, month, year, hour, minute, second)
- a stream type represents all possible 8-bit quantities except null. Its contents are interpreted and evaluated by the receiving entity.

Parameter declarations are used in the signatures of services. Each parameter declaration consists of a name and a type.

Definition 5-6: (Parameter declaration)

A *parameter declaration* is a tuple of the form $(parameter_name, parameter_type)$ where $parameter_type \in T$ and $parameter_name$ is an identifier.

If a service is defined in a REWORK specification, it may be requested by an EOB and some EOB in the system has to generate an event as a reaction conforming to the service specification, i.e., defined as one of the valid events in the service specification.

Invariant 2: If a service is defined in a REWORK specification then a server EOB must exist for it.

Services are defined in a REWORK specification as follows:

Definition 5-7: (Service definition)

A *service definition* is a 5-tuple $sd = (service_name, request_parameters, confirmation, replies, exceptions)$ where

- $service_name$ is a unique identifier (1)
 - $request_parameters$ is a set of request parameter declarations (2)
 - $confirmation$ is an expression of the form $confirm_name$ (3)
 - $replies = \{r_1, \dots, r_n\}$ is a set of *reply definitions*; these are expressions of the form $reply_name (parameter_declarations)$ where $parameter_declarations$ is a set of parameter declarations (4)
 - $exceptions = \{e_1, \dots, e_n\}$ is a set of *exception definitions* which are expressions of the form $exception_name (exception_parameters)$ where $exception_parameters$ is a (possibly empty) list of strings (5)
-

The following clarifications are made to the parts of the above definition:

- (1) Service names are unique within the scope of a REWORK specification.
- (2) Service parameters are bound to values during run-time by the service clients.
- (3) A service confirmation is generated once an appropriate server is assigned.
- (4) Service replies express the possible outcomes of a complete service execution. They are processed by the service client.
- (5) Service exceptions express unexpected situations which may occur during the execution of a particular service. Service exceptions denote that an ad hoc deviation of workflow execution will occur following the service execution which causes the exception. Further kinds of exceptions are discussed below.

For notational convenience in this text, services are defined with the following syntax:

```
service service_name (request_parameters) {
    reply_definition_list
    exception_definition_list
};
```

For example, a service describing the execution of a query over some database system can be specified:

```
service DatabaseQueryService (
    string q_lang;    // the query language in which the query is written
    string q_def ;    // the query expression
    string q_db ) {   // the name of the database which has to be used (if known)
replies:
    QueryResult ( string q_result );
    InvalidDatabase ( string error_code );
    InvalidQueryFormat ( string error_code );
exceptions:
    InoperativeQueryServer ( string q_server );
}; // DatabaseQueryService
```

The scanning of a document is a service required in the mail handling workflow. The service defines no request parameters. Successful service execution provides either an output file of a specific format, or some error code from the scanning application. If the scanning service is not available an appropriate exception event is raised.

```
service ScanDocumentService () {
replies:
    Success ( string file_name, string scan_format );
    Failure ( string error_code );
exceptions:
    NoScannerAvailable ( );
}; //ScanDocumentService
```

Workflow task execution corresponds to service execution by an EOB. The set of available services D thus expresses which atomic steps can be executed in a particular workflow system. Services can be requested by client EOB generating request events in the context of an executing *workflow* as will be described below.

Workflows are auxiliary constructs which are used to group execution of services by giving them a common identifier. A workflow has a type which defines a set of services (and other non-identical workflows) which are to be requested once its execution is initiated. It also defines a set of service replies which signal that its execution has completed. Workflow execution termination results in one of the defined replies for the workflow type. Workflow requests and replies are generated only by the event engine.

Invariant 3: If a workflow type is defined in a REWORK specification, then a server EOB must exist for it.

Invariant 4: If a workflow type references a service or other workflow type, then the service or workflow type must also be defined in the REWORK specification.

The set of all services and workflows in a REWORK specification comprises the set D of service/workflow definitions of a REWORK system.

5.3 Primitive Events in the Workflow System Metamodel

The execution of workflows by EOB is based on their reaction to events. Events represent the occurrence of interesting situations in the REWORK system. They are also the basic integration and interaction mechanism between EOB. In this section we describe in detail the various types of primitive events used for modeling interaction, as well as their intended use, and structure. We distinguish between *event types* defined by *event expressions* and *event occurrences* which are instances of these types.

Primitive events in the REWORK metamodel are typed atomic happenings of interest generated by EOB during workflow execution. The different event types described in this section denote different meanings ascribed to the event occurrence and imply certain qualities of the occurrences for the type:

- the mandatory system-provided attributes of the event occurrence, and
- the permissible additional user-defined attributes.

Primitive event occurrences conceptually belong to three categories: *time events* occur within the workflow system, *interaction events* are signaled by EOB and express the execution state transitions of workflow tasks by EOB, and EOB *internal events* refer to events which are visible only within an EOB. While the first two categories are important for workflow execution, EOB internal events are relevant only for the implementation of EOB and are described in the next chapter. Summarizing:

Definition 5-8: (Primitive event type)

A primitive event expression is a string denoting a time event definition or an EOB interaction event definition. A *primitive event type* is the entity created by the assignment of an identifier to this expression.

The following subsections discuss the different types of primitive events and their use in modeling workflow systems. We base our definitions on the following auxiliary sorts:

- $S = \{s_1, \dots, s_n\}$ — A set of participating sites. Sites are named hierarchically according to the Internet DNS naming scheme.
- $B = \{b_1, \dots, b_n\}$ — A set of defined EOB.
- $W = \{w_1, \dots, w_n\}$ — A set of running workflows (workflow instances).

5.3.1 Time Events

Events are instantaneous occurrences in time—that is they have no duration—consequently every point in time potentially represents a primitive event. In REWORK systems, points in time can be measured in relation to the reading of an imaginary system wide clock—the *global reference clock*. The concept of time [Jensen et al., 1994] which is supported by the REWORK metamodel can be characterized as follows:

- Time is linear, allowing comparison of *timestamps*.
- Time has a *lower bound* coinciding with the workflow system initialization, but no upper bound.

- Each participating site has a local clock out of which a *local site time* can be read. A *global time* is approximated by adjusting the granularity of local clocks to the global reference clock granularity g_g . We assume that g_g —dependent on the synchronization precision among the local clocks—is small enough such that no two primitive events originating at the same site occur at the same global time. This assumption allows us to use a simplified semantic model for timestamps in distributed workflow systems, as suggested in [Schwiderski, 1996], without affecting the power of our model. More specifically, it excludes the concurrent execution of workflow steps at any given site. In the typical application domain of REWORK architectures, this restriction is easily met or can be attained by defining a new site (and clock).
- The three conceptual primitives of time *points* (e.g., 17:00 on 30.5.1998), time *intervals* with a lower and an upper bound (e.g., from this instant until 17:00 on 30.5.1998), and time *durations* (e.g., 24 hours) must be supported. These primitives are expressed as part of the time event type definitions.

Time events in the REWORK metamodel can be absolute, relative, or periodic. *Absolute time events* express real-time points and are defined in terms of a time specification expressing a date and time recorded by a local clock site.

Definition 5-9: (Absolute time event type)

An *absolute time event type* is the entity defined by the assignment of a unique identifier to an expression of the form
 day/month/year-hour:minute:second@site, where site $\in S$.

Thus an example of legal absolute time event type is $E_I = \text{time } 30/5/1998-17:07:40@\text{kornat.ifi.unizh.ch}$.

Relative time events define a positive temporal offset of the form days/months/years-hours:minutes:seconds to another event type called the *reference* event type; the offset has to be related to the time occurrence of the reference event type and thus has to be measured at the same site as the reference event type. An example of a legal relative event type to the reference event type E_I is the event type $E_2 = 4/2/0-11:03:00@\text{kornat.ifi.unizh.ch}$ after E_I which occurs 2 months, 4 days, 11 hours and 3 minutes after E_I based on the time that elapses on kornat.ifi.unizh.ch. *Periodic time event types* identify recurring time events based on the local clock of the given site. They are defined based on the time elapsing between each consecutive occurrence so that $E_3 = \text{every } 0/0/1-00:00:00.0@\text{kornat.ifi.unizh.ch}$ defines an event that occurs once a year after its definition time. Alternatively an initialization time can be given for periodic time events determining their first occurrence, for example, $E_4 = \text{every } 0/0/1-00:00:00 \text{ after } 30/5/1998 \text{ } 17:07:40@\text{kornat.ifi.unizh.ch}$. The set of absolute, relative, and periodic time events defined in an REWORK specification is denoted as TET and the set of the event type identifiers as $TETN$.

5.3.2 Interaction Events

The exchange of coordination information between EOB in a REWORK system takes place by the signaling and reaction to EOB *interaction events*. Thus the semantics of EOB interaction events are associated with execution states of workflows. The manifestation of these events are event messages which are forwarded by the underlying communication infrastructure to EOB which have a registered interest in these events, called the event *subscribers*. Interaction events are service requests, confirmations, replies, and exceptions. These events contain system-provided (implicit) and user-provided parameters. Through the use of *service request events* the exe-

cution of services by EOB can be triggered. The initiation of service execution is signalled by *request confirmation events*. Finally, the results of service execution can be communicated to other EOB by *service reply events*.

Service execution can take place in a *synchronous* or *asynchronous* mode. If it is synchronous, the service client is automatically registered as a subscriber of the confirmation and the reply event. If it is asynchronous, the client may or may not choose to subscribe (and define some reaction) to the confirmation and reply event.

An interaction event occurrence is then explicitly raised by an EOB which plays the role of the *publisher* with respect to the subscribers of this event type. Interaction event types are derived from service definitions as follows:

- A *request event type* is defined for each service definition. The type name is the service name, and the event parameters are derived from the service request parameters. For example, based on the database query service (see above) the following request event type will be defined:
request DatabaseQueryService (string q_lang, string q_def, string q_db);

Invariant 5: If a service is defined in the REWORK specification, then a request event type with the same name and formal parameters is defined.

- A *reply event type* is defined for each reply defined in a service. The type name is composed from the service name to which the reply name is appended. For example, based on the database query service the following reply event types will be defined:
reply DatabaseQueryService.QueryResult (string q_result);
reply DatabaseQueryService.InvalidDatabase (string error_code);

Invariant 6: If a service reply is defined in the REWORK specification, then a reply event type with the name of the form *service_name.reply_name* and the same formal parameters is defined.

- A confirmation event type is defined for each service definition. Service confirmation events are used to inform service clients that a server has assumed the responsibility to execute the service. A confirmation event is generated by a subscriber which decides to execute the requested task prior to the actual task execution. The event type name is composed from the service name to which the string *Confirm* is appended. For example, based on the database query service the following confirmation event type will be defined:
confirm DatabaseQueryService.Confirm;

Invariant 7: If a service reply is defined in the REWORK specification, then a reply event type with the name of the form *service_name* is defined.

- An *exception event type* is defined for each exception defined in a service. Note that additional exception types are predefined in the system (see below). For example, based on the database query service the following exception event type will be defined:
exception DatabaseQueryService.InoperativeQueryServer (string q_server);

Invariant 8: If a service exception is defined in the REWORK specification, then an exception event type with the name of the form *service_name.exception_name* and the same formal parameters is defined.

Formal event parameters are instantiated when an event occurs and assigned values which are transmitted to all event subscribers. Summarizing:

Definition 5-10: (Event parameter)

An event parameter is a pair (*parameter_type parameter_name*), where *parameter_type* $\in T$ and *parameter_name* is an identifier unique for the event type.

We now define EOB interaction event types:

Definition 5-11: (EOB interaction event types)

let *name* be the name of a service/workflow $d \in D$, and *parameter_list* a possibly empty list of typed event parameters, then the expression

- request *name* (*parameter_list*) denotes the service/workflow request event type
- confirm *name*.Confirm denotes the service request confirmation event type
- reply *name*.reply_name (*parameter_list*) denotes the service/workflow reply event type
- exception *name*.exception_name (*parameter_list*) denotes an exception event type

Nothing else denotes an EOB interaction event type.

Figure 5-14 depicts the available primitive event types in the REWORK metamodel. *REQ*, *CFM*, *RPL*, and *EXC* are respectively the sets of request, confirmation, rejection, reply and exception event types defined in a given REWORK specification. $PET = TET \cup REQ \cup CFM \cup RPL \cup EXC$ is the set of primitive event types defined for the REWORK specification. *REQN*, *CFMN*, *RPLN*, *EXCN*, are the corresponding sets of the event type identifiers and $PETN = TETN \cup REQN \cup CFMN \cup RPLN \cup EXCN$.

Given an interaction event type *etype* the following operations is defined:

- service_of : *etype* \rightarrow *service*, where *service* $\in D$

Given a service *service* the following operations are defined:

- request_of : *service* \rightarrow *rtype* $\in REQ$
- confirmation_of : *service* \rightarrow *ctype* $\in CFM$
- replies_of : *service* $\rightarrow REPLIES$, where $REPLIES \subseteq RPL$
- exceptions_of : *service* $\rightarrow EXCEPTIONS$, where $EXCEPTIONS \subseteq EXC$

5.3.3 Exception Events

Exception event types defined in a REWORK specification describe abnormal workflow execution situations. A REWORK system can deal with exactly those semantic failures which can be described in terms of the REWORK metamodel. These failures are expressed by the exception clauses in service specifications. In case of service execution termination one has to distinguish however, between abnormal service termination and undesired service termination. An undesired termination is not an exception as its meaning is that a service provided a reply which indicates that it successfully terminated albeit with an undesired result from the perspective of the applica-

tion. By abnormal termination on the other hand a service execution had to be terminated before it could provide any of the valid results (replies) defined for it. The REWORK metamodel does not deal with *system-generated exceptions*, i.e. exceptions that are caused by errors in the underlying execution platform.

The special meaning of exception event types with respect to the other event types consist in the fact, that for each such type defined, a corresponding exception handling component must be defined, i.e., an EOB which has some predefined reaction on the event occurrence. In the rest of this section we concentrate our discussion on the classification of the kinds of exceptions in the REWORK metamodel:

- *ad hoc exceptions*: they signal the starting point of ad hoc workflow execution. It is the responsibility of the EOB reacting to a service exception to ensure that the workflow will continue execution at the correct point. This means that at some point the exception handler must initiate a subsequent step in the workflow by raising an appropriate request event.
- *otherwise exceptions*: they denote situations which have not been covered in the workflow specification but for which some EOB in the REWORK specification defines exception handling behavior. Due to the fact that a REWORK system includes the description of the WFMS infrastructure, it is possible to specify the handling of situations which are outside the scope of a workflow specification and involve the workflow system infrastructure.
- *true exceptions*: denote situations that are so unanticipated that none of the defined EOB can handle them. True exceptions occur when no EOB in the REWORK system defines a reaction for the particular event type. These kind of exceptions are outside the scope of the REWORK system.

Invariant 9: If an exception type is defined, an event handler EOB must exist for this type in the REWORK system.

The different types of exceptions have to be handled differently at the architectural level. Service exceptions require no special handling as they essentially refer to aspects pertaining to the scope

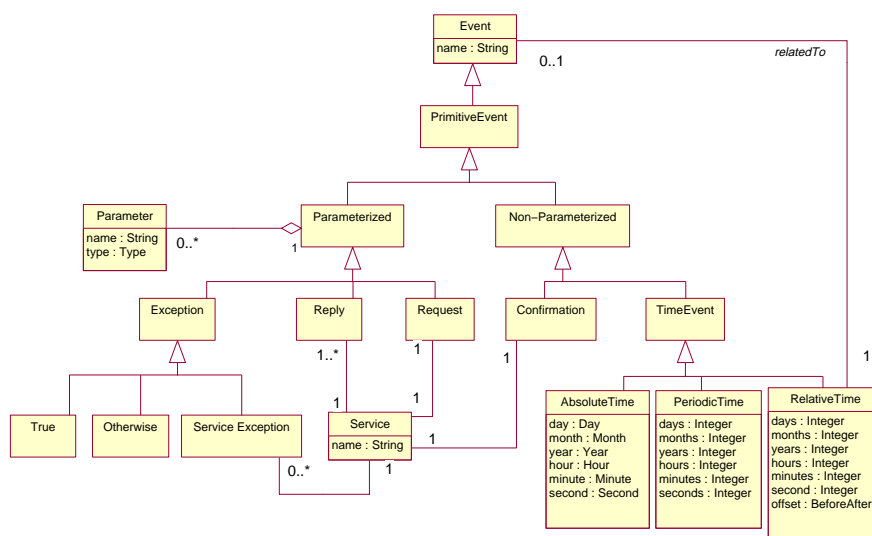


Figure 5-14: Class diagram [Booch et al., 1999] of the primitive event types in the REWORK metamodel.

of the workflow specification. The main implication is that the exception handling EOB must ensure the executing workflow proceeds in a well-defined way. Otherwise exceptions require that some corrective measures have to be defined, probably involving interaction with the original service client or workflow execution abortion. True exceptions cannot be handled within an REWORK system.

5.3.4 System-Controlled Event Occurrence Parameters

As mentioned above, interaction event occurrences contain a number of parameters which are implicitly defined for every interaction event type. These system-controlled parameters are described in Table 5-1. They can be accessed by using the dot (.) operator. For example, the EOB name which generated reqt1 can be accessed as reqt1.origin.

Table 5-1: System-controlled event occurrence parameters for EOB interaction events.

<i>Parameter</i>	<i>Description</i>	<i>Value domain</i>
type	event type	request, confirm, reply, exception
event identifier	site-specific unique identifier	$0..2^{32}-1$
site	occurrence site	S
timestamp	local time at the event occurrence site, i.e., the site of the publisher	time point above lower bound of workflow system time (see below)
name	event name	$PETN$
origin	the EOB which generated the event (the publisher)	B
antecedent	an event occurrence which causes this occurrence	reference to event occurrence or NULL
workflow	workflow instance during which event occurred	W

5.3.5 Timestamps and Primitive Event Occurrences

In order to define the semantics of event occurrences, we introduce the concept of a *timestamp*. As mentioned before, a workflow system is composed of a set of distributed sites S each of which has a single *local* clock which is read by the system when a new event occurrence is generated at this site. Given S and a function $gt: local \rightarrow global$ calculating the global time gt_s of a local clock lt_s at a site s , the timestamp of an event occurrence is defined as follows:

Definition 5-12: (Timestamp)

The *timestamp* $T(e)$ of an event occurrence is a partial function $T(e): S \rightarrow global$ defining a global time $global_s = gt_s(lt_s)$ for each site s participating in the timestamp domain.

Based on the definition of primitive event types and timestamps we define *primitive event occurrences* in a REWORK system.

Definition 5-13: (Primitive event occurrence)

A primitive event occurrence in a REWORK system is a 9-tuple $e = (type, site, eid, T, name, wf, origin, antecedent, parameters)$, where

- $type \in PET$ is the event type of e

- $site \in S$ is the site where e occurred
- $0 \leq eid \leq 2^{32}-1$ is the unique identifier for $site$ of the event occurrence
- $name \in PETN$
- T is the timestamp of e defined as follows:

$$T(e)(site).global = gt_s(lt_s)$$

$$T(e)(s').global = \perp \quad \forall s' \neq site$$
- $wf \in W$ is the workflow instance in which e occurred
- $origin \in B$ is the event publisher EOB
- $antecedent$ is a reference to a further event occurrence
- $parameters$ is a list of actual parameters

Note, that a globally unique event identifier is defined by the tuple $(site, eid)$. Also note that wf , $source$, and $parameters$ are meaningless for time events, since these occur independently from workflows and cannot have parameters. In case of a request occurrence, its informer is the client of the service while in case of confirmations and replies, it is the service provider.

5.4 Composite Events in the REWORK Metamodel

Composite events are used to express complex workflow situations. As with primitive events, we can conceptually distinguish between event types and occurrences of these types. *Composite event types* are defined by applying unary and binary *event operators* on event types. The participating event types, i.e., the event types to which the operators are applied, are called the *component event types* of the composite event type. The definition of new composite event types does not cancel the definition of their component types.

The event operators provided by the REWORK metamodel are similar to those used in centralized ADBS. The main extension is the concurrency operator which is meaningful only for distributed systems. In this section we describe the available composite event operators and the semantics of event composition in a REWORK system. We note the similarity of operators defined here and those described in ADBS literature. Examples of event specification languages in ADBS can be found in [Gatzju, 1995] and [Chakravarthy & Misra, 1994] and the AIDE active information tool-box [Jasper, 1994].

5.4.1 Timestamp Relations

The semantics of composite event occurrences depend on the notion of time used in the distributed workflow system. For this purpose we have to extend the notion of timestamps as defined for primitive event occurrences. Informally, a timestamp determines the occurrence time of an event. While primitive event occurrences have a timestamp which can be directly read from the local clock of the occurrence site and subsequently be mapped to a global time (see Definition 5-12), the timestamps of composite event occurrences must be constructed depending on the type of the composite event. To define the semantics of event composition in a distributed system we use the notion of a component timestamp:

Definition 5-14: (Component timestamp)

A *component timestamp* t of a timestamp $T(e)$ is a tuple $(site, global)$, where $site \in domain(T(e))$ and $global = T(e)(site).global$. We also write that $t \in T(e)$.

Thus, for composite event occurrences, a timestamp $T(e)$ is a complex entity which may contain one or more component timestamps for which the global clock time has been calculated for the different sites of the component occurrences. The addition of new component timestamps depends on the relationship between timestamps of the events being composed (see Definitions 5-15 to 5-17).

A $2g_g$ -restricted temporal ordering between primitive event occurrences based on their timestamps can be established [Schwidderki, 1996]. Thus, in order to determine the temporal order of two primitive events occurring at different sites, their timestamp difference must be at least $2g_g$. Otherwise these events are perceived to occur concurrently. This means that a partial order structure of primitive event occurrences can be defined. Under the assumption that there are no two primitive events occurring at the same site at the same global time, the temporal relationships between two timestamps $T(e_1)$ and $T(e_2)$ are defined as follows:

Definition 5-15: (Concurrent timestamps)

The *concurrency relationship* between timestamps, written as $T(e_1) \parallel T(e_2)$, is said to hold iff

$$\begin{aligned} & \forall t_1 \in T(e_1) \forall t_2 \in T(e_2) : \\ & (t_1.site = t_2.site \wedge t_1.global = t_2.global) \vee \\ & (t_1.site \neq t_2.site \wedge |t_1.global - t_2.global| < 2g_g) \end{aligned}$$

Two timestamps are thus concurrent if all their component timestamps measured at the same site are equal and all their component timestamps from different sites differ less than $2g_g$.

Definition 5-16: (Sequential timestamps)

The *sequential relationship* between timestamps, written as $T(e_1) < T(e_2)$, is said to hold iff

$$\begin{aligned} & \exists t_1 \in T(e_1) \exists t_2 \in T(e_2) : \\ & (t_1.site = t_2.site \wedge t_1.global < t_2.global) \vee \\ & (t_1.site \neq t_2.site \wedge t_1.global < t_2.global - g_g) \wedge \\ & \forall t_1 \in T(e_1) \forall t_2 \in T(e_2) t_1.global \leq t_2.global \end{aligned}$$

Two timestamps are sequential if for any of the component timestamps a precedence relationship can be established. This means that if component events originate at different sites, their difference must be at least two global clock ticks, while if they originate at the same site, they must be at least one global clock tick apart. For every preceding component timestamp the global time is not larger than that of the following component timestamp. Note that the sequential relationship is transitive. Two timestamps for which neither a concurrency, nor a precedence relationship can be established, are said to be unrelated when their base-values —at least one of which is non-atomic— are less than one clock tick apart.

Definition 5-17: (Unrelated timestamps)

Two timestamps $T(e_1)$ and $T(e_2)$ are said to be *unrelated*, written as $T(e_1) \diamond T(e_2)$ iff

$$\neg ((T(e_1) \parallel T(e_2)) \vee (T(e_1) < T(e_2)) \vee (T(e_2) < T(e_1)))$$

The timestamps of composite events are determined by the latest timestamp of a component occurrence. In order to unambiguously determine the composite event timestamp, if there are concurrent or unrelated timestamps, a *timestamp join* procedure is used (see Definition 5-18). Note that if no precedence relationship can be established between the participating timestamps $T(e_1)$ and $T(e_2)$, at most two global ticks cover them.

Definition 5-18: (Joined timestamps)

Given two timestamps $T(e_1)$ and $T(e_2)$ for which neither $T(e_1) < T(e_2)$ nor $T(e_2) < T(e_1)$, their *joined timestamp* is a new timestamp $T(e)$ calculated by the function $T(e_1), T(e_2) \rightarrow T(e_1) \cup T(e_2)$ defined as follows:

$$\begin{aligned} \text{joinset} &:= \text{set}_1 \cup \text{set}_2 \\ \forall s \in \text{joinset } T(e).(s) &:= \begin{cases} T(e_1)(s) & \forall s \in \text{set}_1 \setminus \text{set}_2 \\ T(e_2)(s) & \forall s \in \text{set}_2 \setminus \text{set}_1 \\ \max \{ T(e_1)(s), T(e_2)(s) \} & \forall s \in \text{set}_2 \cap \text{set}_1 \end{cases} \end{aligned}$$

Informally, the joined timestamp is the higher of the local clock values recorded for each participating event occurrence.

5.4.2 Composite Event Restrictions

In many cases, the application semantics require that certain restrictions are imposed on event composition. These are applied during the detection of composite event occurrences. *Composite event restrictions* can be expressed based on the system-controlled parameters of component event types. A general approach towards composite event restrictions is provided in [Schwiderski, 1996], where during composite event type definition the specification of parameters which have to be equal are explicitly defined. In [Gatziau, 1995] the expressiveness of composite event restrictions is contrasted to the effects attained by the comparison of system defined event parameters in ECA-rule conditions. The semantics of the two approaches are different and in some cases the semantics expressed by the restriction cannot be expressed in the condition. Here, we limit our considerations to the specific parameters that are interesting from a workflow specification perspective, i.e., the workflow instance in which an event occurs and the origin of an event occurrence.

Composite events may have to be restricted to describe situations which relate to a particular workflow instance or situations which occur when a particular constellation of events occur in different workflow instances. In order to specify that the component events of a composite event have to occur within the same workflow execution instance, the boolean *same-workflow* restriction may be used in a composite event expression. Note, that *same-workflow* is not meaningful with disjunction events, as the situation expressed by the event occurs in exactly one workflow. Analogously, the boolean *same-broker* restriction can be used on a composite event expression to specify that the composite event shall be detected, only if the component events have the same EOB as their origin. Note again, that *same-broker* restriction cannot be used with disjunction and concurrency event types, as an underlying assumption of the model states that only one event can occur at a particular point in time on one site.

5.4.3 Composite Event Types

Composite event types are defined by applying event composition operators on existing primitive and composite event types. The permissible composite event types are expressions of the form described in the following definition:

Definition 5-19: (Composite event type)

- A primitive event type is a *composite event type*.
 - If E_1, E_2, E_3 and E_4 are composite event types, then
 - $\text{and}(E_1, E_2) : \text{same-workflow:same-broker}$, is a *conjunctive* event type
 - $\text{seq}(E_1, E_2) : \text{same-workflow:same-broker}$, is a *sequential* event type
 - $\text{ccr}(E_1, E_2) : \text{same-workflow}$, is a *concurrent* event type
 - $\text{or}(E_1, E_2)$, is a *disjunctive* event type
 - $\text{xor}(E_2, E_3, [E_1, E_4] : \text{same-workflow:same-broker})$, is an *exclusive disjunctive* event type
 - $\text{iter}(E_1, E_2) : \text{same-workflow:same-broker}$, is an *iterative* event type
 - $\text{not}(E_1, [E_2, E_3] : \text{same-workflow:same-broker}) : \text{same-workflow:same-broker}$, is a *negation* event type
 - $\text{rep}(E_1, n) : \text{same-workflow:same-broker}$, is a *repetitive* event type
 are composite event types, where the restrictions *same-workflow* and *same-broker* are optional composite event restriction expressions, and $n \in \mathbb{Z}^+$.
 - Nothing else is a composite event type.
-

We define an operation $\text{component_types}(E)$ on event types which returns the set of participating component event types—including the event type itself. CET is the set of all composite event types and $CETN$ the corresponding set of type identifiers. $ET = PET \cup CET$ is the set of all event types defined in the system. Note that composite event types whose component types relate to different sites are considered as *global*. For notational convenience, composite event types can be defined explicitly with the following syntax:

define event event_name = *composite event expression*

The event name can be reused in the definition of other composite event types or be referenced in the event part of EOB ECA-rules (see below). Composite event types can also be defined implicitly in the event clause of an EOB ECA-rule definition similar to the language of the SAMOS active database system [Gatziau, 1995].

As can be assumed from definition 5-19, nested composite types may be defined by subsequent application of multiple event composition operators on resulting event types. We note however, that not all nested composite event expressions are meaningful. More specifically, combinations of unary operators and combinations of binary and unary operators must be analyzed. For example, the repetition of a negation or the sequence of two negations do not make sense. This issue is discussed extensively for events in SAMOS in [Gatziau, 1995].

5.4.4 Semantics of Event Composition

We can now define the semantics of composite event occurrences as determined by the type of these occurrences and eventual event restrictions. These semantics depend on the notion of time used in the distributed workflow system. A composite event occurrence is detected at a site in the workflow system which is called the *event detection site*. The location of this site depends on the underlying event detection mechanisms. A composite event occurrence is then defined as follows:

Definition 5-20: (Composite event occurrence)

A composite event occurrence is a 6-tuple $ce = (type, site, eid, T, components)$, where

- $type \in CET$ is the event type of ce
 - $site \in S$ is the *detection site* of ce
 - $0 \leq eid \leq 2^{32}-1$ is the unique occurrence identifier for that detection site
 - T is the timestamp of ce
 - $components$ is a list of *component event occurrences* with elements of the form $(site, eid)$.
-

Note that, as in primitive occurrences, a globally unique event identifier is defined by the tuple $(site, eid)$. A composite event occurrence is considered to have its workflow identifier and publisher built from the respective identifiers of its component events (see Table 5-3). When composing events which are composite occurrences in themselves, the *same-workflow* and *same-broker* restrictions consequently refer to set equality. Furthermore, we assume the following in a complete REWORK system:

Invariant 10: For defined each event type there exists exactly one component in the REWORK system which is able to detect occurrences of the type.

Henceforth, EO is the sort of (primitive and composite) *event occurrences*. In order to define the subset of CET of actually occurring composite events we use the concept of a (global) event occurrence sequence:

Definition 5-21: (Event occurrence sequence)

An *event occurrence sequence* eos , is a finite sequence of n primitive and/or composite event occurrences. eos is written as $\langle e_1, \dots, e_n \rangle$, where each e_i , $1 \leq i \leq n$ is an occurrence of some event type at some point in global time.

It is also valid that $\forall i, k, 1 \leq i, k \leq n : (T(e_i) < T(e_{i+k})) \vee (T(e_i) \parallel T(e_{i+k})) \vee (T(e_i) \Diamond T(e_{i+k}))$, i.e. eos represents a partial ordering among the participating event occurrences with respect to their occurrence timestamps.

We can now define the semantics of the various event composition operators based on the occurrences of events of the appropriate types. All eligible event occurrences used for event composition are consumed in the chronicle consumption mode. We thus assume the following lemma is valid for all subsequent definitions:

Lemma: Let CE be a composite event type, eos an event occurrence sequence, and E_1, E_2, E_3 , and E_4 event types. Then CE has an occurrence ce with component event occurrences $ce.components = \langle e_1, \dots, e_n \rangle$ iff

$$\forall_E e : e \in ce.components \Rightarrow \neg \exists CET ce' : ce \neq ce' \wedge ce' \in eos \wedge e_i \in ce'.components$$

The *conjunction* operator and (E_1, E_2) is used when both component events E_1 and E_2 are to occur, irrespective of their relative order of occurrence. E_1 and E_2 may originate from the same or from different EOB and sites. A conjunction event occurrence is assigned the timestamp of the later of the component occurrences, if their temporal order can be decided. Otherwise, the two timestamps have to be joined. This is formally expressed as follows:

Definition 5-22: (Conjunction)

CE is a conjunction occurrence iff:

$$\begin{aligned}
 CE &= \text{and } (E_1, E_2) : \text{same-workflow:same-broker} \wedge \\
 &\exists_{E1} e_1 \exists_{E2} e_2 : ce.components = \langle e_1, e_2 \rangle \wedge \\
 &e_1 \in eos \wedge \\
 &e_2 \in eos \wedge \\
 &T(ce) = T(e_2) \text{ iff } T(e_1) < T(e_2) \wedge \\
 &T(ce) = T(e_1) \text{ iff } T(e_2) < T(e_1) \wedge \\
 &T(ce) = T(e_1) \cup T(e_2) \text{ iff } (T(e_1) \parallel T(e_2) \vee T(e_1) \diamond T(e_2)) \wedge \\
 &e_1.wid = e_2.wid \text{ iff same-workflow} = true \wedge \\
 &e_1.origin = e_2.origin \text{ iff same-broker} = true
 \end{aligned}$$

The *sequence* operator $\text{seq}(E_1, E_2)$ denotes that a "happened before" relation exists between the two component event occurrences. The operator is used when a predefined order has to be imposed over the occurrence of events. Depending on whether the component events occur at the same site or not, the semantics of sequence are different. In any case, the sequence occurrence is assigned the timestamp of the later event occurrence.

Definition 5-23: (Sequence)

CE is a sequence occurrence iff:

$$\begin{aligned}
 CE &= \text{seq}(E_1, E_2) : \text{same-workflow:same-broker} \wedge \\
 &\exists_{E1} e_1 \exists_{E2} e_2 : ce.components = \langle e_1, e_2 \rangle \wedge \\
 &e_1 \in eos \wedge \\
 &e_2 \in eos \wedge \\
 &T(e_1) < T(e_2) \wedge T(ce) = T(e_2) \wedge \\
 &e_1.wid = e_2.wid \text{ iff same-workflow} = true \wedge \\
 &e_1.origin = e_2.origin \text{ iff same-broker} = true
 \end{aligned}$$

The *exclusive-or disjunction* operator applied on events E_2 and E_3 $\text{xor}(E_2, E_3, [E_1, E_4])$ has a meaning only within a defined interval (written as $[E_1, E_4]$). Thus, either an E_2 or an E_3 occurrence but not both, must occur within the interval defined by E_1 and E_4 and a sequence of an E_1 and E_4 occurrence must exist. An exclusive-or disjunction occurrence is signalled only after the interval expires, i.e., an occurrence E_4 occurs and is assigned the timestamp of the component event that actually occurred. The same-workflow and same-broker restriction can be required for the interval definition. This is expressed as follows:

Definition 5-24: (Exclusive-or disjunction)

CE is an exclusive-or disjunction occurrence iff:

$$\begin{aligned}
 CE &= \text{xor}(E_2, E_3 [E_1, E_4] : \text{same-workflow:same-broker}) \wedge \\
 &((\exists_{E2} e_2 : ce.components = \langle e_2 \rangle \wedge e_2 \in eos \wedge T(ce) = T(e_2) \wedge T(e_1) < T(e_2) < T(e_4) \wedge \neg \exists_{E3} \\
 &e_3 : e_3 \in eos \wedge T(e_1) < T(e_3) < T(e_4)) \vee \\
 &(\exists_{E3} e_3 : ce.components = \langle e_3 \rangle \wedge e_3 \in eos \wedge T(ce) = T(e_3) \wedge T(e_1) < T(e_3) < T(e_4) \wedge \neg \exists_{E2} \\
 &e_2 : e_2 \in eos \wedge T(e_1) < T(e_2) < T(e_4))) \wedge \\
 &e_1.wid = e_4.wid \text{ iff same-workflow} = true \wedge \\
 &e_1.origin = e_4.origin \text{ iff same-broker} = true
 \end{aligned}$$

The *inclusive-or disjunction* operator $\text{or}(E_1, E_2)$ means that the event occurrence is detected as soon as one of the component events occurs. An inclusive-or disjunction occurrence is signalled immediately after on eof the component events occurred and is assigned the timestamp of the component event that actually occurred. This is expressed as follows:

Definition 5-25: (Inclusive-or disjunction)

CE is an inclusive-or disjunction occurrence iff:

$$\begin{aligned} CE &= \text{or} (E_1, E_2) \wedge \\ &(\exists_{E_1} e_1 : ce.components = \langle e_1 \rangle \wedge e_1 \in eos \wedge T(ce) = T(e_1)) \vee \\ &(\exists_{E_2} e_2 : ce.components = \langle e_2 \rangle \wedge e_2 \in eos \wedge T(ce) = T(e_2)) \end{aligned}$$

The *concurrency* operator $\text{ccr} (E_1, E_2)$ is used when both component events are to occur virtually at the same time. Note, that a basic assumption underlying the REWORK metamodel is that the component occurrences of a concurrent event can only occur at different sites. It is not possible to establish a temporal order between them. The timestamp of the concurrent event occurrence is derived from the timestamps of both component events through the join procedure (see Definition 5-18). Thus:

Definition 5-26: (Concurrency)

CE is a concurrent occurrence iff:

$$\begin{aligned} CE &= \text{ccr} (E_1, E_2) : \text{same-workflow} \wedge \\ &\exists_{E_1} e_1 \exists_{E_2} e_2 : ce.components = \langle e_1, e_2 \rangle \wedge \\ &e_1 \in eos \wedge \\ &e_2 \in eos \wedge \\ &T(e_1) \parallel T(e_2) \wedge T(ce) = T(e_1) \cup T(e_2) \wedge \\ &e_1.wid = e_2.wid \text{ iff same-workflow} = \text{true} \end{aligned}$$

An *iteration* operator $\text{iter} (E_1, E_2)$ is used to define that all occurrences of an event E_1 are to be collected *until* the occurrence of an event E_2 . The iteration event occurrence is assigned the timestamp of the delimitation occurrence event of type E_2 . At least one occurrence of type E_1 is necessary for the detection of the iteration. Note that an iteration event type where an occurrence of type E_1 does not have to occur before an occurrence of type E_2 is also conceivable.

Definition 5-27: (Iteration)

CE is an iterative occurrence iff:

$$\begin{aligned} CE &= \text{iter} (E_1, E_2) : \text{same-workflow} : \text{same-broker} \wedge \\ &\forall i, 1 \leq i \leq n \exists_{E_1} e_i \exists_{E_2} e_{n+1} : ce.components = \langle e_1, \dots, e_n, e_{n+1} \rangle \wedge \\ &T(ce) = T(e_{n+1}) \wedge \\ &\forall k, 1 \leq k \leq n+1 : e_k \in eos \wedge \\ &e_1.wid = e_k.wid \text{ iff same-workflow} = \text{true} \wedge \\ &e_1.origin = e_k.origin \text{ iff same-broker} = \text{true} \end{aligned}$$

A *negation* operator $\text{not} (E_2, [E_1, E_3])$ is a special case of a sequence, where the sequence event occurs only if there has been no occurrence of E_2 in the interval defined by E_1 and E_3 . The timestamp of the occurrence is that of the component occurrence defining the end of the interval, i.e., E_3 .

Definition 5-28: (Negation)

CE is a negation occurrence iff:

$$\begin{aligned} CE &= \text{not} (E_1, [E_2, E_3] : \text{same-workflow-1} : \text{same-broker-1}) : \text{same-workflow-2} : \text{same-broker-2}) \wedge \\ &\exists_{E_2} e_2 \exists_{E_3} e_3 : ce.components = \langle e_2, e_3 \rangle \wedge \\ &e_2 \in eos \wedge \\ &e_3 \in eos \wedge \\ &T(ce) = T(e_3) \wedge \end{aligned}$$

$$\begin{aligned}
&e_2.wid = e_3.wid \text{ iff } \text{same-workflow-1} = \text{true} \wedge \\
&e_2.origin = e_3.origin \text{ iff } \text{same-broker-1} = \text{true} \wedge \\
&(\neg \exists_{E1} e_1 : e_1 \in eos \wedge T(e_2) < T(e_1) < T(e_3) \wedge \\
&e_1.wid = e_2.wid = e_3.wid \text{ iff } \text{same-workflow-2} = \text{true} \wedge \\
&e_1.origin = e_2.origin = e_3.origin \text{ iff } \text{same-broker-2} = \text{true})
\end{aligned}$$

A *repetition* operator $\text{rep} (E_1, n)$ defines a further special case of a sequence, where the precedence relationship is valid among each pair of consecutive component occurrences. Its meaning is that a repetition event occurs as soon as the component event has occurred n times, where n is a positive integer. The timestamp of the composite occurrence is that of the last component occurrence.

Definition 5-29: (Repetition)

$$\begin{aligned}
&CE \text{ is a repetitive occurrence iff:} \\
&CE = \text{rep} (E_1, n) : \text{same-workflow} : \text{same-broker} \wedge \\
&\forall i, 1 \leq i \leq n \exists_{Ei} e_i : ce.components = \langle e_1, \dots, e_n \rangle \wedge \\
&T(ce) = T(e_n) \wedge \\
&\forall k, 1 \leq k \leq n : e_k \in eos \wedge \\
&e_1.wid = e_k.wid \text{ iff } \text{same-workflow} = \text{true} \wedge \\
&e_1.origin = e_k.origin \text{ iff } \text{same-broker} = \text{true}
\end{aligned}$$

The detection of composite event occurrences is an implementation issue described in more detail in chapter 9. At this point it suffices to say that composite event occurrences are detected at the site where the corresponding detector object resides. The detection is synchronous and based on the assumption that all relevant component event occurrences with smaller timestamps will have arrived at the detector (*FIFO* delivery).

5.4.5 System-Controlled Composite Event Parameters

Analogously to primitive event occurrences, composite event occurrences have various system-controlled parameters, which are defined in Table 5-2. Note that composite events in general do not have a unique origin or common workflow identifier.

Table 5-2: System-controlled event occurrence parameters for composite event occurrences.

Parameter	Description	Value domain
type	event type	and, or, seq, ccr, not, iter, rep
timestamp	occurrence timestamp	see above
site	detection site	S
components	array of component event occurrences	elements are event occurrences

5.4.6 Composition of Event Occurrence Parameters

A final issue which has to be considered for composite events is what happens to event parameters when event occurrences are composed, i.e., which parameters of the component event occurrences are available—and consequently can be referenced—in the composite event occurrence. This depends on the composition operator as is illustrated in Table 5-3.

In conjunction, sequence, and concurrent events the parameters of both component occurrences are available. In disjunction events the parameters of the occurred event are available. In iteration

Table 5-3: Composition of event parameters. We assume that an event occurrence e_1 of type E_1 and e_2 of type E_2 are composed.

Composite event occurrence	Valid parameters
and (e_1, e_2)	$\text{parameters}(e_1) \cup \text{parameters}(e_2)$
or (e_1, e_2)	$\text{parameters}(e_1) \mid \text{parameters}(e_2)$
xor ($e_1, e_2, [\text{interval}]$)	$\text{parameters}(e_1) \mid \text{parameters}(e_2)$
seq (e_1, e_2)	$\text{parameters}(e_1) \cup \text{parameters}(e_2)$
ccr (e_1, e_2)	$\text{parameters}(e_1) \cup \text{parameters}(e_2)$
iter (e_1, e_2)	$\cup_n \text{parameters}(e_i) \cup \text{parameters}(e_2)$
not ($e_1, [\text{interval}]$)	none
rep (e_1, n)	$\cup_n \text{parameters}(e_i)$

events all parameters of E_1 occurrences and the delimiting event occurrence are available. In repetition events all parameters of E_1 occurrences are available. Finally, in negation events no parameter of the component event is available.

5.5 Event History in a REWORK System

In an executing REWORK system event occurrences are collected in an *event history*. The notion of event history is formalized and forms the foundation for the operational semantics of EOB, which are elaborated on in the next chapter. Based on the previous definitions we can define the event history of the system execution at time t as follows:

Definition 5-30: (Event history at global time t , EH_t)

EH_t is an event occurrence sequence with n occurrences such that for all $e \in EH_t$ holds

- $T(e) < t - \text{max_sync_delay}$
- $\exists E \in ET : e.type = E$

$\exists m, m \geq 1 : \forall i, 1 \leq i \leq m : \exists E_i \in ET : \exists_{E_i} e_i : \exists CE \in CET : \exists eos :$

$(e_i \in EH_t \wedge eos = \langle e_1, \dots, e_{i-1}, e, e_i \dots e_m \rangle \wedge eos \text{ occurs_for } CE \wedge$

$(\forall_{E'} e' : e \neq e' \wedge e' \in EH_t \wedge$

$\langle e_1, \dots, e_{i-1}, e', e_i \dots e_m \rangle \text{ occurs_for } CE \Rightarrow T(e) < T(e'))$

$\Rightarrow \exists_{CE} ce : ce \in EH_t \wedge e \in e.components)$

Informally, an event history contains all global event occurrences in the REWORK system which occurred up to the last detector synchronization point ($t - \text{max_sync_delay}$). All occurrences in the event history have a corresponding event type defined. Furthermore, if an occurrence can form a composite event occurrence together with other occurrences, then a corresponding composite global event occurrence must also be contained in the event history. Whenever a composite event occurrence is built, then for each of its component events the oldest eligible candidate is chosen, i.e., according to the chronicle parameter context [Chakravarthy et al., 1994].

The event history EH_{t+1} will be constructed by appending to EH_t all primitive and composite event occurrences with timestamp $t+1$. The order of the appended events is not important as there are no hidden time dependencies defined in the workflow within the appended occurrences. These new composite event occurrences are determined as described in definitions 5-22 to 5-29.

5.6 Summary

In this chapter we introduced the connectors provided by the REWORK metamodel used for the description of the architecture of workflow systems. These are services describing REWORK system functionality, and events describing the way REWORK system components may interact. The computational components of the REWORK metamodel are described in the next chapter.

Component interaction is based on the exchange of asynchronous messages representing event occurrences in the REWORK system. Workflows are executed by the reaction of the workflow system components to these event occurrences. The reaction includes the generation of further events and results in an event history which represents the execution of the workflow by the participating components. The operational semantics of primitive and composite events in an REWORK system are defined in the next chapter based on the notion of a correct event history.

6 Reactive Workflow System Components

The explicit definition of the architecture of a workflow system provides advantages for the composition of workflow systems and the subsequent analysis of their properties. In this chapter we complete the description of the REWORK metamodel by describing the provided component abstractions. The elements of the REWORK metamodel that we consider include the following:

- the computational components which compose a REWORK system;
- the functionality provided by each component, i.e., its computational content (see Definition 2-5);
- the composition and configuration relationships between REWORK components;
- the actor integration mechanisms provided by REWORK components; and
- the mechanisms for the choice of appropriate components for the execution of given tasks, i.e., task assignment specification.

The chapter is structured as follows: in section 6.1 we generally characterize event occurrence brokers (EOB). In the subsequent section we describe the characteristics of the different types of EOB provided by the REWORK metamodel. In section 6.3 we turn our attention to the definition of organizational relationships in REWORK systems. In section 6.5.1 we consider task assignment. We conclude the chapter by defining the correctness of EOB behavior over resulting event histories, thus formalizing the operational semantics of workflow execution by EOB in a REWORK system.

6.1 Event Occurrence Brokers

A workflow system consists of heterogeneous *actors*. Each actor has its own notion of the real world and of how it interacts with it. The realization of this notion can be manifested in various interaction mechanisms which are expressed in the actor connotation (see chapter 4). For example, a database server understands statements written in some database manipulation language, while a person understands words in some natural language. The REWORK metamodel provides the concepts and specification constructs required for the definition of a homogeneous workflow system architecture, thus avoiding the architectural mismatch expected during the actor integration in a workflow system. Actors are represented by composite reactive components called *event occurrence brokers (EOB)*. Actors are considered to execute workflow tasks as a reaction to workflow-relevant situations in their environment (e.g., a particular moment in time has arrived, or a previous task has been completed). The occurrence of these situations is manifested by REWORK events. EOB generate and react to REWORK events, and map their meaning to concepts present in the vocabulary of the respective actor. In other words, they provide a homogeneous representation of heterogeneous real-world actors and provide the actor functionality to the workflow system-internal mini-world.

In the REWORK metamodel the main essence of an EOB is that of an intermediary. Specifically, an EOB acts as an intermediary between the rest of the REWORK system and a particular real-world application or human user. The main conceptual properties of EOB are summarized here:

- An EOB is an instance of a non-abstract aggregate component defined in the REWORK metamodel. The composer of a workflow system deals with EOB during its composition.
- An EOB is a software entity which executes workflows as a result of an autonomous reaction to event occurrences captured by its computational interface from its environment.
- An EOB represents real world actors. It transforms REWORK events to interactions which are defined in the application vocabulary and provokes application processing in response to situations manifest in the REWORK system.
- Each EOB is composed of other components which implement the various properties and functionality of the actor's connotation in the workflow system. The internal structure of an EOB is *invisible* to other EOB in the REWORK system. The internal EOB components are subtypes of generic component types customized to the functionality needed to represent a particular actor.

The situations that may arise in the REWORK system are described by primitive and composite events, as described in the previous chapter. The behavior of EOB is expressed by ECA-rules which define which events interest the EOB and what its reaction is to occurrences of these events. The reaction patterns defined by ECA-rules either belong to the standard behavior of the EOB—its predefined reactive behavior—or can be assigned dynamically as *rule packages*. Rule packages describe how the default behavior of an EOB has to be modified for a specific workflow or a specific period in time.

6.2 Actor Representations: The REWORK Components

In this section we describe the typology of components provided by the REWORK metamodel and the ways in which actors can be represented with the provided component types. In general, top-level workflow system components can be of one of the following non-abstract types which are the leafs of the REWORK component type hierarchy: external components—meaning that the entities they represent are external to the (REWORK) workflow system—organizational units, and internal reactive components (EOB) which are part of the workflow system and execute the workflow tasks and supporting services. In general, REWORK system component instances are aggregate objects whose type is defined in the REWORK metamodel. Thus, instantiating operations for the types are defined in the metamodel for the non-abstract types.

Organizational units represent real world organizational entities such as company departments, or project teams. Organizational units can have organizational relations defined between them and other user-defined descriptive properties. Their descriptive properties are defined by named character strings and serve only documentation purposes. Two subtypes of organizational units, users and groups, are defined in sections 6.2.4 and 6.2.5. Organizational relationships are defined below. Group and user components inherit the properties of both organizational units and EOB. Furthermore, REWORK system sites are objects of the site type. Figure 6-1 depicts the RE-

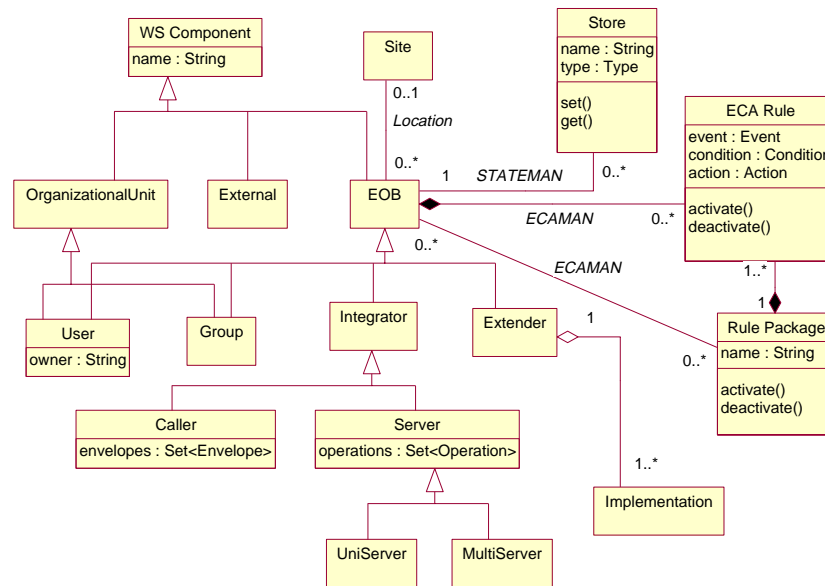


Figure 6-1: A class diagram [Booch et al., 1999] of the REWORK metamodel component type hierarchy and related elements.

WORK metamodel component type hierarchy and related types in the metamodel. These are described in detail in the subsequent sections.

6.2.1 Workflow System-External Components

External components provide the stimuli which arrive from outside the system (e.g., the initiation of a workflow activity) by declaring the service that they request. Note, that this service must be provided by some component in the system. External components do not provide services to other components, but are used to initiate the workflow execution from the outside world. They are used to define the interfaces between real-world entities which are not an integral part of the workflow system.

An external component represents, for example, the mail system which delivers a document in the mail handling workflow of the case study. This component actually initiates a new instance of the workflow by requesting the ScanDocument service and is defined as follows:

```

external MailDelivery {
    requests ScanDocument ();
};
  
```

6.2.2 Workflow System-Internal Components

Workflow system-internal components —called EOB— represent actors which participate in workflows by requesting and providing the defined services. During their operation, EOB are always assigned to a specific site in the workflow system, they can however be either connected or disconnected to this site.

Each EOB has a unique identifier in a REWORK specification. It consists of various subcomponents which communicate with each other by exchanging events over an EOB-specific internal message bus. The type of the EOB determines what properties these subcomponents have, and what events they understand. However, every EOB has at least the following (sub)components: an event delivery (EDI) and an event posting (EPI) component, a persistent state management com-

ponent, an ECA-rule management component, and an actor management component. The structure of EOB is depicted in Figure 6-2.

Invariant 11: Each EOB in a REWORK system consists of an internal message bus, an event delivery, an event posting, a persistent state management, a rule management, and an actor management component. The properties of these components must be defined in the system.

Internal Event Messages

The components of an EOB communicate by the broadcasting of messages over a reliable shared message bus (component (1) in Figure 6-2) which provides a single message broadcast operation. Thus, all components connected to the message bus are notified of an internal message and react accordingly. The set of messages each component understands is fixed for its type. The general EOB-internal message structure is the following: (message_name, message_origin, message_parameters). The message parameters are typed with their types defined in OMG Interface Definition Language [OMG, 1995]. Messages can be sent either in a synchronous or asynchronous mode; in the first case the message sender blocks until it receives a response message with the output parameters, in the second case the message sender can continue processing. The behavior of an EOB subcomponent when it receives a message is defined as part of the subcomponent interface.

State Manager

An EOB contains a *state manager* (STATEMAN, component (2) in Figure 6-2) implemented over some storage system not further specified. The STATEMAN is responsible for the persistent storage of the state objects. The persistence is achieved by reachability from a system-provided persistent root whose component objects and their attributes are declared in the EOB state.

The STATEMAN manages typed objects which are used for the persistent storage of workflow data. The permissible object types are a subset of the ODMG ODL [Cattell et al., 1997] type system. For example, a scanned document type can be defined as follows:

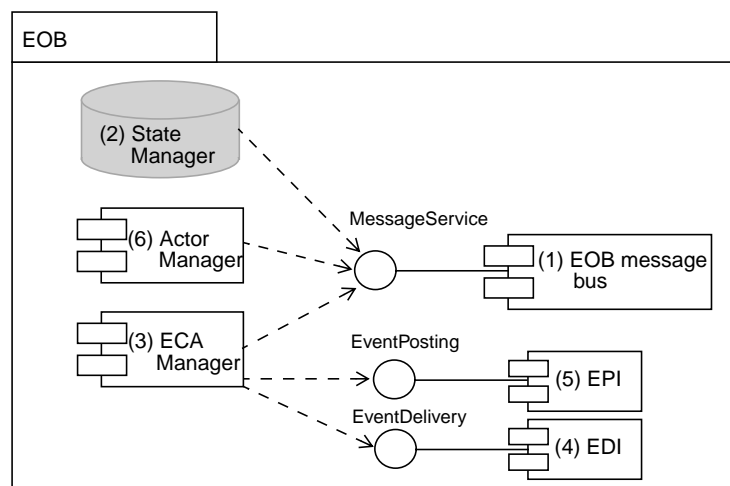


Figure 6-2: A component diagram [Booch et al., 1999] of the internal structure of an EOB. Circles denote component interfaces and dashed arrows denote dependency relations. Component numbers (1) to (6) refer to explanations in the text.


```

struct TimeAndDate(date dt, time tm) ;
type ScannedDocument {
    attribute string file_name;
    attribute TimeAndDatecreated;
    attribute TimeAndDateupdated;
    attribute string scan_format;
};

```

The definition of state objects implies that for each of their attributes an assignment message `set()` and a retrieval message `get()` is understood by the STATEMAN. These messages are handled as synchronous call/return operations, i.e., the generating subcomponent of the EOB is guaranteed to receive a response message by the STATEMAN.

EOB Behavior

Every EOB includes an *ECA-rule manager* (ECAMAN, component (3) in Figure 6-2) responsible for management and execution of ECA-rule objects (or simply ECA-rules) expressing the behavior of the EOB. In general, the ECA-rules are used to provide the following functionality:

- *Subscription of the EOB to the corresponding event type*: once a particular ECA-rule is defined for an EOB and until it is explicitly deactivated (see below), the EOB is registered in the system as a subscriber to the event type contained in the event clause of the rule.
- *Execution of services by actors*: in general, REWORK service specification does not have to correspond to the functionality provided by actors. It may be the case for example, that a REWORK service is implemented by a series of queries in a database and the subsequent evaluation of the service results. This processing logic is described in ECA-rules which bridge the semantic gap between application operations and REWORK services.
- *Enforcement of task execution ordering*: a workflow specification expresses various dependencies among workflow tasks. These dependencies are expressed by composite events and by the rule actions which generate new events which eventually trigger the subsequent tasks. The mapping of execution ordering to rules is described in the next chapter.
- *Guarding task execution conditions*: ECA-rule conditions can be used to express task execution constraints which can be evaluated against task execution results available through event parameters.
- *Exception and failure handling*. ECA-rules are used to specify failure handling. They can also express recovery policies as mentioned previously.

The general structure of the behavioral rules in EOB is the following:

```

define rule rule_name follows rule_name {
    on event_expression
    [if condition_expression]
    do action_definition
};

```

The operational semantics of these rules corresponds to those of ECA-rules in active database systems: the occurrence of an event of a given type referenced in the rule event clause causes the rule to fire, consequently the condition is checked and if this evaluates to true the action of the rule is performed. The defined set of rules in an EOB forms its *ruleset*.

The conditions are expressed over the EOB state, i.e., involve accessing the value of some state object through the STATEMAN, as well as over the parameters of the event occurrence.

The action definition is a set of statements which defines the reaction of the EOB to the event and eventually generates new EOB interaction events which are communicated to the rest of the REWORK system. The statements in the actions are essentially accesses to the connectors provided by the other EOB subcomponents.

Rule object execution can be selectively activated and deactivated by the ECAMAN through a corresponding `activate(ECA_rule)` and `deactivate(ECA_rule)` operation for it. By using this operation, the association between an event and an EOB action can be temporarily switched off. By default, rules are activated until the deactivation operation is defined; it remains in the deactivated state until `activate` is called for it. The rule deactivation mechanism is usually used when situation-specific *rule packages* are assigned to an EOB which have to override its predefined behavior. The REWORK metamodel provides the rule package construct for defining sets of rules which are manipulated as a whole. The rule package is a named container object for ECA-rules which understands the following messages:

- `insert (ECA_rule)` adds a rule object to the rule package;
- `delete (ECA_rule)` delete a rule object from the rule package;
- `activate (for_EOB)` activate the rules in the package for the EOB;
- `deactivate (for_EOB)` deactivate the rules in the package for the EOB.

The relative execution order of rules can be defined during EOB specification. Rules can be partially ordered by specifying relative execution ordering, as for example supported in SAMOS [Gatzju, 1995].

ECA Manager

Rule execution on the type of the EOB. In general, a rule is executed by the ECAMAN component which is responsible for the following functions:

- consumption of incoming events from the EDI of the EOB,
- detection of primitive and composite events,
- scheduling of rule execution according to defined relative priorities and subsequent rule execution,
- rule object administration (i.e., activation and deactivation), and
- posting of corresponding events to the REWORK system via the EPI and depending on the internal processing of the EOB.

The ECAMAN encapsulates the other EOB subcomponents from the REWORK system. It is aware of the EOB-internal message format used over the EOB message bus, as well as of the REWORK event format and communication protocol.

New REWORK events to which the EOB has subscribed arrive at its *event delivery interface* (EDI, component (4) in Figure 6-2). The EDI is a persistent message-queue to which incoming events are appended by an atomic `put()` operation and are consecutively consumed by the ECAMAN of the EOB by means of an atomic `consume()` operation. Once an event is consumed, the ECAMAN must determine which of the defined ECA-rules is triggered by the incoming occurrence. The event detection mechanism is considered in more detail in chapter 10. It suffices to note, that the formal semantics of event composition—as defined in the previous chapter—have to be respected.

EOB interaction events generated during rule execution are reinserted in the REWORK system by the ECAMAN through a persistent outgoing FIFO queue which is called the *event posting interface (EPI)* depicted by (5) in Figure 6-2. As explained below, writing to the EPI occurs within the transaction that the rule is executed.

Rule execution by the ECAMAN takes place in sequential mode. Rule processing is performed according to the following steps (see also the collaboration diagram [Booch et al., 1999] in Figure 6-3):

- (1) An event arrives at the delivery interface of the ECAMAN.
- (2) At some point in time, the ECAMAN process examines the oldest event in the delivery queue, by sending EDI the message 2.1 : peek(), and determines the set of active rules in the rulebase that are fired by this event (2.2 : getRule()). Subsequently fired rules are inserted into a persistent *pending evaluation queue (PEVQ)* by 2.3 : putRef(). The relative order of the rules in the queue depends on the defined rule priorities. If no rule priorities are defined, the queueing order depends on the specification order or rule creation timestamp (starting from the oldest rule first).
- (3) The conditions of the rules in the queue are evaluated sequentially possibly involving interaction with the STATEMAN. For each rule whose condition is true, the action will have to be executed. A reference to these rules is inserted in a persistent *pending execution queue (PEXQ)*. Rules whose conditions are false are removed from the PEVQ. The event is in any case removed from the EDI by sending it a consume() message.
- (4) Each rule in the PEXQ is subsequently examined as follows: if the event is a request, and the rule action produces a reply event, then a confirmation event is 4.3 : put() to the EPI. This actually guarantees that a reply or exception must be generated by the EOB at some later time. We note, that although only a single rule can be defined in each EOB which generates a reply, multiple rules may be triggered by the same request without however, producing a subsequent reply. The actions of the rules in the pending execution queue are executed sequentially one-by-one. After action execution is completed (i.e., 4.4 : exec() returns), each rule is removed from the PEXQ and the action of the next rule in the queue is

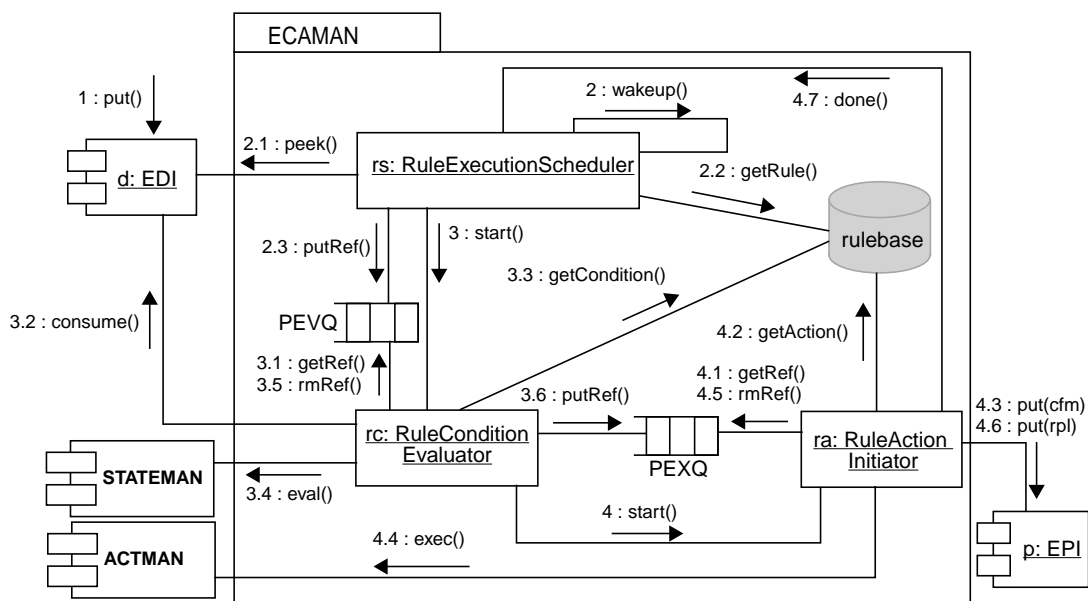


Figure 6-3: Collaboration diagram [Booch et al., 1999] of event and rule processing by the ECAMAN.

retrieved. If an action produces a reply event, this is 4.6 : put() on the EPI. When the PEXQ is empty, the next event can be examined from the delivery interface.

Note that steps 2 and 3 are executed within one ACID transaction. Furthermore, the actions of each rule in the PEXQ are executed in separate transactions; at the end of each transaction the reference to the rule is removed from the PEVQ (messages 4.4 and 4.5). Thus, while the coupling mode [McCarthy & Dayal, 1989] between event detection and condition evaluation is *immediate*, the coupling mode between condition evaluation and action execution is *sequential causally dependent* [Buchmann et al., 1995].

We note, that no nested rule execution, as for example in active database systems, can occur in the sequential mode because the events generated by rule actions are immediately posted to the REWORK system. They will be consumed during the next rule execution cycle after their subsequent arrival to the EDI. An increased efficiency in event processing and rule execution can be achieved by introducing multiple rule action executor threads. Each action part of the ECA-rule can be executed in a separate thread. This is only meaningful if the represented actor supports concurrent processing. Additionally, from the perspective of the REWORK system, the effects of action execution—reads and writes from the STATEMAN—must be serializable and respect the rule priority ordering. For a discussion of transaction serialization issues we refer to the relevant database literature (e.g., [Bernstein et al., 1987]).

Processing Entity Manager

Interaction with actors is implemented by the *actor manager* (ACTMAN, component (6) in Figure 6-2). In general, the ACTMAN implements the access to the functionality of actors by using some wrapping technique to access the external systems; it may alternatively implement the desired functionality itself in some programming language by directly accessing the functionality provided underlying REWORK system infrastructure. The ACTMAN however provides a set of operations which are called in ECA-rule actions. Its implementation properties correspond to the automation and connector type attributes of the actor connotation.

As already mentioned, the ACTMAN may wrap an external system or implement system functionality. When the ACTMAN directly implements desired functionality, the REWORK system infrastructure plays the role of a white-box actor for a workflow task. The implementation of different wrapping techniques is a problem which has been addressed by a large body of research; in our work we concentrate on the issues relevant to integrating these wrapping techniques in a reactive component-based architecture. Some details on this interesting issue are considered in the following sections.

6.2.3 Types of EOB

We now describe the different types of EOB provided by the REWORK metamodel. As previously mentioned, EOB are constructed by using of one of the predefined types as a specification template. The set of defined EOB types in a REWORK specification is denoted as *EOBT*. The typology is determined by properties of the ACTMAN and ECAMAN subcomponents of an EOB as depicted in Table 6-1.

Table 6-1: The EOB typology as determined to the subcomponent properties.

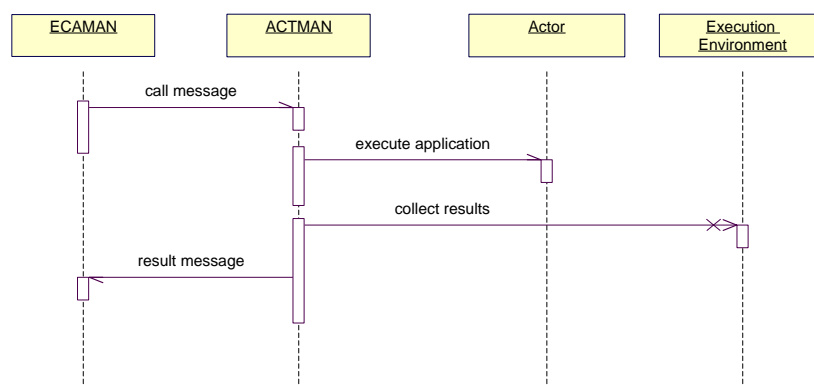
<i>EOB type</i>	<i>ECAMAN rule action execution</i>	<i>ACTMAN functionality</i>
caller	single-threaded	execution of envelope
uni-server	single-threaded	implementation of a wrapping method for an external server system
multi-server	multi-threaded	implementation of a wrapping method for an external server system
user	single-threaded	user worklist management and interaction
group	single-threaded	–
uni-extender	single-threaded	implementation
multi-extender	multi-threaded	implementation

Callers

Callers represent actors whose execution is asynchronous from the perspective of the REWORK system. This essentially means that the termination of a service execution has to be recognized by the EOB, as the external program does not provide any feedback on its execution termination with respect to the results of the service execution. Examples of such programs include non-interactive applications such as various UNIX tools but also interactive applications which only indicate their termination status as successful or unsuccessful.

As already mentioned, an actor can provide different interaction mechanisms expressed by its connector type attribute. The connector types supported by callers include batch, operating system call, script, and shared data. In all cases, the result of the service executed by the actor must be retrieved and evaluated by the EOB. This has implications for the processing of service requests, as well as the implementation and interaction of caller subcomponents.

In a caller, each ECA-rule whose action produces a reply event is an envelope [Dowson, 1987] executed by the ACTMAN. The ACTMAN may invoke the external application, i.e., the actor, by an operating system call (e.g., UNIX `exec`), call script language commands, or submit a batch file with the appropriate parameters. It then blocks until the application completes execution and subsequently collects the eventual results and returns them to the ECAMAN. Depending on the connector type, in some cases, the application may signal its execution termination by returning an exit code, while in other cases, the ACTMAN may have to poll a file system for the ex-

**Figure 6-4:** Sequence diagram [Booch et al., 1999] of interaction between subcomponents of a caller, the actor and its execution environment.

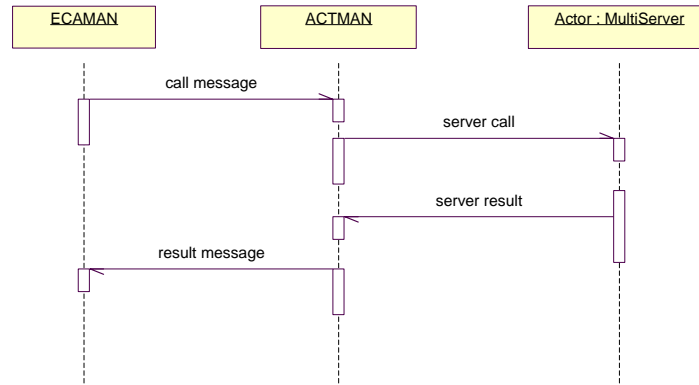


Figure 6-5: Sequence diagram [Booch et al., 1999] of interaction between the subcomponents of a multi-server and the actor.

istence of a file. Different kinds of ACTMAN/actor interaction have to be implemented in the caller to support each case.

Note that the ECAMAN may call multiple ACTMAN operations within the execution of a single ECA-rule. Each of these operations may result in the execution of a different envelope. In that case, the intermediate results must be collected by the STATEMAN. The interaction patterns between the subcomponents of a caller EOB are depicted in Figure 6-4.

Servers

Server EOB represent actors which are configured and operated as server applications. This has implications on the implementation of the ACTMAN, which must be able to establish a connection to the server application, call an operation and receive the results. Furthermore, the type of server and the way the server can be accessed influences the properties of the ECAMAN. Servers are either multi-servers or uni-servers. *Multi-servers* represent server applications which concurrently service multiple requests (e.g., typically a database server) while *uni-servers* represent applications which can service a single request at-a-time (e.g., a printer spooler). While this property is usually not interesting from a workflow specification perspective, it is important for the implementation of the ACTMAN, and in addition, may influence the efficiency of execution of workflow tasks. During market-based workflow execution (as explained in the chapter 7), the type of server can be taken into consideration to determine the choice of the actor assigned to a particular task.

Uni-servers are conceptually very simple. During rule action execution, the ECAMAN calls an operation from their interface and the ACTMAN maps this operation to an actor API call or similar interaction mechanism. While the server application services the operation, the ACTMAN blocks and waits until servicing ends. Subsequently the results are transformed to a form understandable to the REWORK system and sent by an EOB message to the ECAMAN rule execution subcomponent.

This basic interaction pattern is similar in multi-server EOB (see Figure 6-5). However, the ACTMAN of multi-servers are more complex as they have to process concurrently multiple operation invocations. Thus, each executing ECA-rule action part may potentially lead to a separate operation call of the server application. The ACTMAN has to maintain different operation contexts for each server invocation. We note, that it is the responsibility of the EOB developer to ensure, that the partitioning of rule actions results in a serializable operation execution. For example, consider a workflow service *s* which is implemented by calls to operations *op1*, *op2*, and *op3* of the represented server actor. A multi-threaded rule execution allows the partitioning of the re-

quest processing in three rules whose event is request(s). However, if the order of execution of the three operations influences their outcome (e.g., they operate on common data), then this order must be maintained during rule execution, which in general excludes their concurrent execution. There are however cases, when concurrent operation execution is desirable, as for example, when op1, op2, and op3 are unrelated database queries whose results are to be collected for the workflow service execution.

6.2.4 Users

In the REWORK metamodel *users* provide the interface to “human actors”, i.e., they represent people participating in workflow execution. The internal organization of user EOB is similar to that of the previously described EOB types. There are however, some special aspects which have to be considered. A user EOB is a subtype of an organizational unit. As such, it may be involved in defined organizational relationships to other organizational units (see below). These relationships are evaluated during task assignment.

The ACTMAN component of a user implements the actual person/machine interface. Different kinds of user interfaces are conceivable, as for example, window-based applications, terminal-based interfaces, or web-browsers. The type of interface and the display platform determine the wrapping functionality required by the ACTMAN. However, every ACTMAN supports the following predefined set of messages:

- *enqueue (service request event)* is provided in order to post the parameters of the requested service to the user interface;
- *dequeue (service request event)* is provided to remove previously posted service requests from the user interface.

Additional messages understood by the ACTMAN of a user EOB may be defined to call additional functionality of the wrapped worklist interface.

The STATEMAN component of the user EOB is responsible for the management of a persistent list of pending requests. Each time the *enqueue()* message is sent over the EOB message bus, the STATEMAN inserts the new request event in the list. When the *dequeue()* message is sent it removes the request event from the list. If a crash occurs, the STATEMAN issues an *enqueue()* message for each event in its queue when it recovers.

The ECAMAN of the user EOB must be informed by the ACTMAN about two EOB-internal events that regard task execution by a person. The first one refers to the acceptance or rejection of task execution by the person. It is signaled by the messages *accept (service_request_event)* or *reject(service_request_event)* respectively. The acceptance or rejection of a task generates a confirmation event or a system-defined exception event respectively, which is subsequently posted in the REWORK system through the EPI.

The second internal event refers to the completion of a manual task signaled by the message *completed(service_request_event, completion_state)*, where the completion state must be one of the reply types that are defined for the requested service. In reaction to this message a service reply event is generated by the ECAMAN and posted over the EPI to the REWORK system. The interactions between worklist components during the execution of a task is displayed in Figure 6-6.

Different interaction scenarios occur if the task executed by a user (lets say *eob₁*) is not manual, but is performed with the help of some actor (e.g., a software system). This actor will be represented by an other EOB (lets say *eob₂*) in the REWORK system. The acceptance of a task will

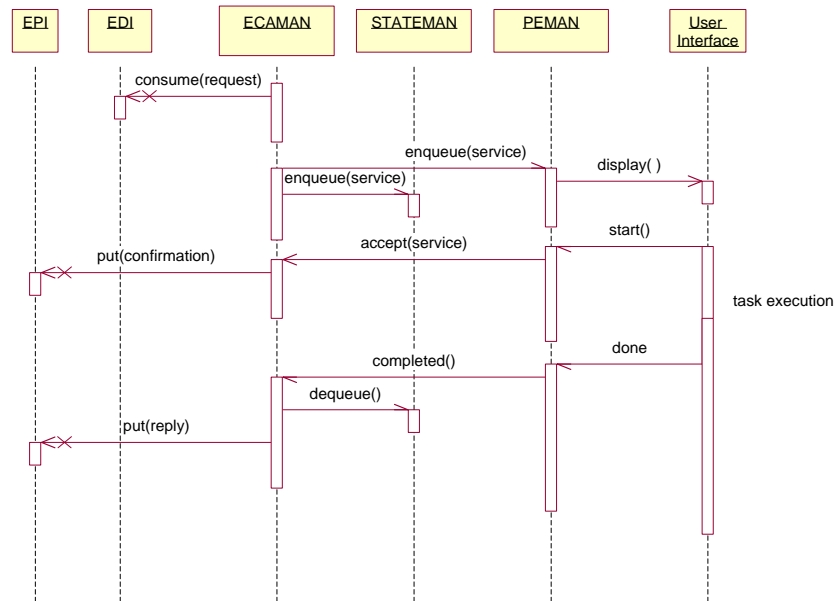


Figure 6-6: Sequence diagram [Booch et al., 1999] of interaction between the subcomponents of a worklist and the user interface during accepted task execution.

create a service request event which will interest eob_2 . Upon completion of the service execution, a reply event will be posted in the REWORK system by eob_2 which will be detected by eob_1 and result in dequeuing the request. Alternatively, eob_1 may decide to reject an assigned task. In that case, the ACTMAN will generate a rejection event which results in the posting of a corresponding otherwise exception event by the EOB to the REWORK system. This otherwise exception must be handled by an appropriate handler which may decide to reissue a request or abort workflow execution.

The implications for the specification of the EOB is that appropriate rules must be defined for the different possible user interactions. The rules refer to workflow request and reply events on the one hand, and ACTMAN events on the other hand. Their exact specification depends on the required interaction model with the user. From the above discussion two alternative scenarios exist for tasks executed interactively:

- the application executes synchronously with respect to the service execution by the worklist user as a requested service: in that case a rule triggered by the reply must be defined for the user EOB, which will evaluate the application results and generate an appropriate reply to the original service request.
- the application executes asynchronously: in that case, the reply to the original service request is generated immediately after the task has been selected for execution. The results of the application execution are evaluated elsewhere in the REWORK system.

6.2.5 Groups

Groups are EOB which represent organizational groups. A group is also a specialization of an organizational unit for which a persistent state and reactive behavior can be specified. A group has a membership as expressed by a set of organizational relationships (see below) to other organizational units. Group members can belong to more than one group at a time. The special property of a group is that its state is directly accessible in read-only mode to its members, i.e., not through the ECAMAN. In other words, it is a shared data store for groups of EOB. Direct state variable

access is possible however, only in read mode; the group EOB defines the rules which cause a state variable update.

Groups are used to implement common storage areas in workflow systems. Examples of groups from the mail handling workflow include the post office, and the groups of insurance clerks which make up service centers. The post office group has a persistent state containing a variable which stores a scanned document. When the scan service successfully completes, the group EOB stores the results in its state, making it accessible to all its members:

```
group PostOffice {
  store ScannedDocument sc_doc;
  define rule StoreScan {
    on reply (ScanDocument.Success, fname, format )
    do sc_doc.file_name.set(fname);
      sc_doc.scan_format.set(format);
      sc_doc.created.set(now);
  };
};
```

Group EOB are consequently used to actively store and update data which is available to multiple REWORK system components. In this sense, they do not encapsulate a real-world actor but are a WFMS component.

6.2.6 Extenders

In addition to the various actors which perform workflow tasks, a workflow system consists of various infrastructure components which implement the needed functionality for workflow management. These are usually subsumed under the generic term of a WFMS. In the WfMC reference model, such components may belong to the build-time environment (e.g., workflow specification editors), to the administration environment (e.g., a monitoring tool), or to the workflow engine itself (e.g., a workflow log component). These components are usually not defined within the scope of the workflow specification. The architectural perspective of the REWORK meta-model however, allows the consideration of such actors as part of a REWORK system, effectively paving the way for the *extensibility of core workflow management system functionality*.

In a REWORK system, services can be used to describe extensions to the basic WFMS functions. These services are then executed by explicitly defined WFMS components called *extenders*. In other words, through the use of extenders the workflow management system functionality is unbundled in components which provide supplementary workflow management services required only in specific application scenarios. The mechanism has effects analogous to O_2 [Babaloglu & Marzullo, 1993] *service-based tool* integration in the software development environment SPADE [Bandinelli et al., 1996]. There the functionality provided by the environment can be extended by the definition of new tools which operate directly on top of the underlying persistent object storage system.

The interesting issue with respect to the other EOB is the additional flexibility provided with respect to rule action specification in ECAMAN. In the types of EOB considered before, the ACTMAN is used to provide an abstraction of the interface(s) provided by actors and make them accessible to the rest of the REWORK system. The actions in ECAMAN rules are thus restricted to calling these operations (by generating EOB internal events), packing the results, and generating the appropriate REWORK system events. In contrast, extenders can define rule actions in any programming language supported by the workflow system infrastructure. This effectively means that they extend the passive functionality of the underlying system with active functionality.

Extender EOB are used to customize the basic WFMS functionality as required by a specific workflow application system. Similar to servers, they can be either single-threaded or multi-threaded allowing the concurrent execution of multiple rules. As in other extensible systems, extender EOB may be advantageous for profiling against competitive products by providing specialized functionality depending on the system configuration, or be used to accommodate changing requirements in the workflow management domain. Extenders are also useful to implement ad hoc actor integration. In chapter 7 we describe the use of an extender EOB to implement market based workflow execution. Further examples of extenders include a workflow execution monitor providing a monitoring service as a separate workflow, and tools of the composition environment such as a component browser.

6.2.7 The Role of the Event Engine in REWORK Specifications

As mentioned in section 5.1.2, event-based integration in REWORK systems is achieved by means of a distributed glue system called the event engine (EVE). In the context of the REWORK metamodel, EVE can be considered as a special-purpose EOB of which an instance is defined by default in every REWORK specification. This has the following implications for the workflow system composer:

- In every REWORK specification there exists an EOB called EVE. EVE does not represent a particular actor but rather the kernel WFMS infrastructure.
- State variables can be defined for EVE just like for other EOB. However, the state of EVE is accessible to—and thus can be used in rules of—every other EOB. Thus EVE is the appropriate component in which global state may be defined.
- Rules can be defined for EVE and rule packages can be assigned to EVE. These rules then have access to the global state.

The special role and implementation of EVE as an event composition and distributed coordination system is described in detail in chapter 9.

6.3 Organizational Relationships

The REWORK metamodel provides a general-purpose mechanism for the representation of *organizational relationships*. Organizational relationships are typed objects representing directed binary associations between pairs of organizational units. The result of using organizational relationships is the generic representation of organizations as a graph with typed nodes being organizational units and edges representing their relationships. An organizational relationship has a source and a target organizational unit (see Figure 6-7). Organizational relationship objects can be created either during system specification or dynamically during workflow execution.

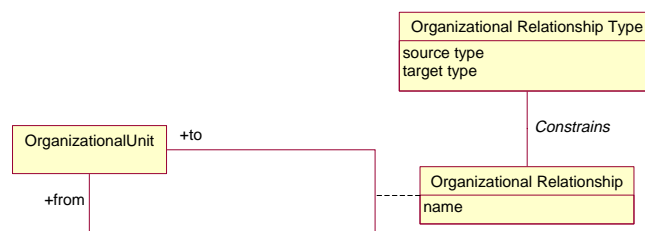


Figure 6-7: Class diagram [Booch et al., 1999] of organizational relationships in the REWORK metamodel.

The advantage of using a generic approach to organizational modeling is that only minimal assumptions are made about the types of organizations which are involved in workflow systems. This compares favorably to approaches which provide only predefined organizational schemas, for example, hierarchies. Leu [Graw & Gruhn, 1995] has a simple schema consisting of persons, roles, and positions, where roles are sets of permissions and positions describe manager relations. In FlowMark [Leymann & Roller, 1994] a distinguishing between organizations, persons, organizational levels, as well as roles is made. A set of fixed roles (manager, coordinator, system administrator) is defined. Provided relationships are substitution, organization relation, manager or person, coordinator, parent organization, level assignment, and role relation. COSA workflow [COSA, 1998] provides a hierarchical organization structure and uses the role concept as a mechanism to build groups of persons. TriGS_{flow} [Kappel et al., 1998] has a schema based on company, department, agent, and role and distinguishes between human and software agents. The TriGS_{flow} allows roles to be specialized providing a simple schema extension mechanism used in task assignment.

Our approach is comparable to the Mobile system [Bussler, 1997] of which we outline the basic characteristics. Two representational constructs are provided which are specified on a type and an instance level: organizational objects and organizational relationships. *Organizational objects* either represent real world entities (e.g., person, machine, server process) or virtual elements (e.g., department, role, group). Organizational relationships can be defined between organizational objects, such as “plays role”, “administers”. Organizational objects and relationships can have attributes defined by their types which are used for task assignment with the definition of agent selection policies. Compared to Mobile, we restrict the specific relationships by constraining the types of objects (organizational units) that can participate in a given relationship type while in Mobile organizational relationships are pairs of strings.

In REWORK relationship types constrain the types of organizational units that can be associated by an organizational relationship. Various *organizational relationship types* are provided by the REWORK metamodel; additional relationship types may be defined during system specification. Predefined relationship types express the kinds of relationships typically encountered in organizations and include:

- relationship type *member_of* (*organizational_unit*, *group*), denoting that the target is a group and the source can be any organizational unit which is its member. Note that nested organizational structures can be expressed by such relationships.
- relationship type *supervisor_of* (*user*, *user*), denoting that the source and targets are user EOB.
- relationship type *representative_of* (*user*, *user*), denoting that the source and targets are user EOB and that the target is equivalent to the source with respect to service execution.

We provide the following definition of organizational relationship types in the REWORK metamodel. Assuming that *OU* is the set of organizational unit EOB defined, then:

Definition 6-1: (Organizational relationship type)

An organizational relationship type *ORT* is a 3-tuple $ORT = (name, source_type, target_type)$, where *name* is a unique identifier, $source_type \in OU$, and $target_type \in OU$.

In the mail handling workflow, for example, the company (TheCompany) is composed of three departments: two service centers and a central post office. Furthermore the person name Johnson works for the post department. Each organizational unit is represented by an REWORK system component. These facts are described as follows:

```

relationship member_of PO_to_Co (CentralPost, TheCompany);
relationship member_of SC_to_Co (SC1, TheCompany);
relationship member_of SC_to_Co (SC2, TheCompany);
relationship member_of Johnsons_Dept (Johnson, CentralPost);
We can formally define organizational relationships as follows:

```

Definition 6-2: (Organizational relationships)

OR is the powerset of organizational unit sets. It constructed by the application of the operation: relationship: $OU \times OU \rightarrow 2^{OU \times OU}$

Two EOB b_1 and b_2 are associated by the organizational relationship *orgrel* iff $orgrel \in OR \wedge (b_1, b_2) \in orgrel$.

In other words, an organizational relationship is a named pair containing two organizational unit EOB. Organizational relationships of a particular type must form acyclic directed graphs.

Invariant 12: The source and target organizational units in an organizational relationship must be defined in the REWORK system.

As already mentioned, organizational relationship objects can be created both during REWORK system specification and during REWORK system operation. Dynamically created organizational relationships are considered for further processing (e.g., in task assignment explained below) immediately following their creation.

6.4 Task Assignment in REWORK Systems

In general, several EOB might be able to react to a certain event (e.g., a request). *Task assignment* is the procedure that determines the EOB registered for that event type, and then chooses those which then will actually be notified of the event occurrence.

There are two possibilities concerning task assignment. If *fixed assignment* is defined during system specification, a certain activity is always performed by a specific actor. In that case a N:1 relation exists between service and executing EOB. Fixed task assignment leads to better performance but may cause interruptions in workflow execution if the assigned EOB is not available for some reason (e.g., is disconnected). Furthermore, the specification of control flow may become overly complex, when many special cases have to be considered. If *flexible assignment* is used, a service is not directly bound to an EOB but is indirectly associated during specification to a representation of one or more EOB from which a suitable EOB is determined at run-time. The functionality provided by an EOB is not a sufficient criterion for servicing a particular request. Flexible task assignment is required in the following cases:

- Task assignment is dependent on properties of the task to be executed. For example, insurance claims with large amounts must be accepted by a staff member which has the right to accept such amounts. Note, that the semantics are different to those of specifying that different activities have to be performed according to the amount (i.e., an XOR-split), as in every case the same activity is performed.
- An organizational relationship may have to exist between the requestor and the provider of a service. For example, a travel expense reimbursement must be signed by the supervisor of an employee.

- A dependency exists between consecutive service executions. For example, the staff member that assumed the processing of an insurance claim and requested a re-scan of an illegible claim document must subsequently resume the processing of the insurance claim.
- A representation may be required if an actor is not available for a limited period. This may for example be the case, if leaves of absence are managed in the workflow system for participating users. Thus, depending on the time when a service is requested, the service providing EOB may be different. Note, that this is not a specification issue but rather a dynamic property of the workflow execution.

Informally, task assignment proceeds in three steps:

- (1) The set of EOB who can potentially react to the event is determined —the set of *subscribers* for the event. It contains all EOB which have rules whose event clause corresponds to the type of the actual occurrence.
- (2) This set can be restricted based on properties such as organizational relationships and the previous execution history. This restriction is provided by a filtering function over capable EOB returning a set of *suitable* EOB.
- (3) One or more EOB from this set are chosen according to run-time properties. The event in question is then dispatched to them. These compose the set of *responsible* EOB for the occurrence.

In the first step, every EOB whose event registration set contains the event type in question belongs to the set of subscribers. Subscription is a static property of EOB. For instance, we require that for each request event type there exists at least one subscriber in the system.

Suitability

In some situations there might be only one subscriber for a specific event in question. Thus, no further task assignment has to occur. In other situations, however, an EOB must fulfill further conditions to be actually suitable, i.e., not every subscriber should actually be notified in a given workflow execution. Hence, suitability is a dynamic property of EOB, which depends on (parts of) the previous workflow execution history, and is expressed over a variety of EOB properties specified when the REWORK system is defined. These properties refer to:

- *event occurrence attributes*: the properties of event occurrences are used to determine which EOB is notified for an event. These may include the timestamp of the occurrence, the origin, and values of the event parameters.
- *EOB properties*: the properties of the EOB are used to determine if it is notified for an event. The properties depend on the type of EOB. Firstly, the EOB may be connected or disconnected to the REWORK system at any given time. Secondly, user EOB have a property defining if they are in a leave of absence and the user that substitutes them during that time. Additional properties are discussed in chapter 8 where market-based workflow execution is described.
- *organizational relationships*: a certain organizational relationship must be defined between the event origin and the notified EOB, e.g., in some request-reply pair, the EOB capable of executing the service must be a supervisor of the requestor.

Suitability is expressed by filter formulas, which can refer to the attributes of an event occurrence, predefined organizational relationships, and properties of EOB. The set of suitable EOB thus includes those subscriber EOB for which the filter condition holds. In filter formulas expressed over event types attributes of instances of the type (i.e., event occurrences) can be referenced. The following filter formula can be defined to express the fact that the group manager must handle a spe-

cial insurance claim case (service `HandleSpecialCase`) in which the staff does not have the appropriate expertise:

suitability HSC (`HandleSpecialCase`, `supervisor_of (origin)`);

For a complete workflow execution, we require that each application of a filter function yields a non-empty set of EOB. Otherwise, the event history contains occurrences to which some EOB *should* react but none is available that actually *will* react. For instance, that could mean that there are requests which cannot be served in the current system configuration—and thus a corresponding exception event must occur. Since suitability is a dynamic property and refers to event occurrences, it cannot be guaranteed at specification time.

Dispatching

While suitability determines the set of EOB to be notified of an occurrence, it still needs to be clarified which ones should react depending on event and workflow type semantics. Concretely, it may be required that only one EOB should react to an event, or that all suitable EOB should react. Thus in this final step called *dispatching*, an event occurrence is finally sent to a (sub)set of the determined suitable EOB. In case only one EOB should react, dispatch yields a singleton set. Dispatching can take place according to different strategies, specified as formulas over event types and applied over event occurrences. Defined strategies are the following:

- *all*: the event is dispatched to every suitable EOB;
- *random*: the event is dispatched to an arbitrary eligible EOB;
- *round robin*: the event is dispatched to that EOB to which an occurrence of the same type was least recently dispatched;
- *load-based*: the event is dispatched to the EOB with the smallest current load, i.e., the smallest number of event occurrences in its delivery interface; and
- *combined*: a prioritized combination of the previous options.

For example, an administrative task such as preparing a standardized letter can be assigned to the member of the staff which currently has the least amount of work to do.

Formal Definitions

For the following definitions we assume that a set B of EOB has been defined. To capture task assignment procedures formally, we introduce a number of abstractions. A subscription maps event types in general and service requests in particular to EOB, meaning that each member of the resulting set can react to instances of the event type—and in case of service requests, is capable of serving the request. Informally, this means that an EOB is a subscriber if it has the event type in its event subscription set, i.e. it has rules triggered by occurrences of the event.

Definition 6-3: (Subscription)

Assume an EOB b and an event type ET are defined. Then the subscription function $\text{subscription}: ET \rightarrow 2^B$ is defined as follows:
 $\text{subscription}(ET) = \{b \mid \exists r \in b.RULES: ET = r.on_etype\}$

Assuming an event occurrence eo , we can define a *suitability function* as follows:

Definition 6-4: (Suitability function)

A suitability function defines a set of suitable EOB by restricting the set of subscriber *EOB* for an event occurrence. Its signature is defined as follows:

$$\text{suitability: } eo \times 2^B \rightarrow 2^B$$

The set of all defined suitability functions is denoted by *SF*. We can furthermore define a dispatch function for the event occurrence as follows:

Definition 6-5: (Dispatch function)

A dispatch function defines a set of dispatch EOB by restricting the set of suitable EOB for an event occurrence. Its signature is defined as follows:

$$\text{dispatch: } eo \times 2^B \rightarrow 2^B$$

The set of all defined dispatch functions is denoted by *DF*. Note that there might be multiple suitability and dispatch functions defined for one event type applied to instances of the type with specific attribute values.

Based on these functions, we determine which EOB are notified of a certain event occurrence. In case the occurrence type is a request, we say that the notified EOB are *responsible* for servicing the request. Informally, we first determine capable EOB and then apply suitability and dispatch functions. Formally, the notification set *NS* of an event occurrence *eo* are defined as follows:

Definition 6-6: (Notification set of event occurrence)

$$NS_{eo} = \{b \in B \mid \exists \text{ suitability} \in SF \exists \text{ dispatch} \in DF: b \in \text{dispatch}(\text{suitability}(eo, \text{subscription}(eo.type)))\}$$

We can thus formally define an REWORK (workflow) system as a set of instantiated EOB, event types which are event occurrences factories, organizational relationships, and task assignment rules, i.e., suitability formulas and dispatch functions.

Definition 6-7: (REWORK system)

A REWORK system *A* is a 5-tuple $A = (B, ET, OR, SF, DF)$, where *B* is a set of EOB, *ET* is a set of event types, *OR* is a set of organizational relationships, *SF* is a set of suitability formulas, and *DF* is a set of dispatch functions.

6.5 Event History in a REWORK System

An executing REWORK system creates an event history, as defined in the previous chapter. The set of request events act as initiating events for a set of workflow executions and thus are referred to as the *input* to the REWORK system.

Definition 6-8: (Event history under REWORK system)

Let *A* be a REWORK system (B, ET, OR, SF, DF) and *eos* be an event occurrence sequence $\langle eo_1, \dots, eo_n \rangle$. The event history EH_t is said to be constructed by *eos* under *A* if it is an event history according to Definition 5-19 and $\forall eo \in eos \Rightarrow eo \in EH_t$.

Typically, we require from a REWORK system that it is able to handle all event occurrences. We thus say, that an event history is *complete with respect to task assignment* if for each event occurrence in the event history a responsible EOB exists, i.e., the notification set of the occurrence is not empty. Thus, the semantics of a REWORK system can be described in terms of event histories: they are defined as the set of event histories which are constructed by it, given a finite sequence of events as input. This implies the notion of event history correctness: a history EH is correct under A if there exists an event occurrence sequence eos , EH is constructed by eos under A , and EH is complete with respect to task assignment.

6.5.1 Operational EOB Semantics

We can now define the semantics of EOB-based workflow execution under a certain event history. We may then define those event histories in which an EOB demonstrates *observably* correct behavior. In general, we note that the operational semantics of an EOB are defined by the events it generates, i.e., for a given EOB, its semantics in terms of events is the restriction of the event history to those occurrences whose origin is the EOB:

Definition 6-9: (Operational EOB Semantics)

The operational semantics of an EOB b under an event history EH are defined as an event history EH' such that

$$(\forall e \in EH': e \in EH \wedge e.origin = b) \wedge (\forall e \in EH: e.origin = b \Rightarrow e \in EH')$$

An EOB is considered a black box with observable external behavior expressed by the events it generates and by two interfaces it provides. It is notified upon the occurrence of those events for which it is in the notification set and reacts according to its definition possibly generating new events. We assume that the event delivery system is reliable, i.e. that every EOB in the notification set of the occurrence will be notified.

Definition 6-10: (EOB as black box)

An EOB $b \in B$, is defined by a 3-tuple $EOB = (RULES, EDI, EPI)$

- $RULES = \{r_1, \dots, r_n\}$ is a set of rules which are pairs of the following form: $(on_etype, genset)$: $on_etype \in ET$, $genset = \{et_1, \dots, et_n\}$: $et_i \in PET$
 - EDI is the *event delivery interface* for incoming event occurrences. Given eo_in , eo_out event occurrences, EH an event history, we define the following operations for EDI :
 $put : EDI \times eo_in \rightarrow EDI'$
 $peek : EDI \rightarrow EDI \times eo$
 $consume : EDI \rightarrow eo_in \times EDI'$
 $flush : EDI \times EH \rightarrow EDI' \times EH' \times eo_out$
 $contains : EDI \times t \times e \rightarrow boolean$
 - EPI is the *event posting interface* for outgoing event occurrences. Given eo_out an event occurrence and EH an event history we define the following operations for EPI :
 $post : EPI \times eo_out \rightarrow EPI'$
 $get : EPI \times EH \rightarrow EPI' \times EH' \times eo_out$
-

The ruleset $RULES$ of the EOB defines its behavior in reaction to incoming events —hence the characterization of an EOB as a *reactive component*. The set of all (primitive and composite) event types to which the EOB has subscribed is called the *event subscription set* $ESSET = \cup_r \in RULES r.comptype(etype)$ of an EOB. When a particular event arrives at the EDI the appropriate rules are fired and this may subsequently result on further primitive events to be generated, i.e., exactly those which are in the *genset* of the fired rule. Thus the $RULES$ defined for a particular EOB determine a dependency between events arriving at the EOB EDI and events posted from

the EOB EPI. The use of the EDI and the EPI is described in section 6.2.2. The operations are required for the definition of the correctness of EOB execution as described in section 6.5.2.

Finally, we note that based on the above definitions we can define correctness criteria over the set B of EOB defined in a given REWORK system A . We can for example require that for a particular primitive event type ET a subscriber to this event is defined. This is expressed as follows:

$$ET \in PET \Rightarrow \exists b \in B_A: ET \in ESSET(b)$$

Similarly other static properties of a REWORK system can be expressed with the use of the existential and universal quantifiers.

6.5.2 Observable Correctness of EOB

Based on the definition of EOB and REWORK systems—including task assignment—we can elaborate on the correctness of EOB. We do not consider the internal behavior of EOB, require however, that an EOB reacts to events of which it is notified and that it generates the appropriate events in response. In other words, we treat an EOB as a black box and only consider the events “going in” and “coming out” of the EOB. We therefore define the *observable behavior of EOB*.

Given the formal notion of event history, we can reason about the correctness of observable EOB behavior. Informally, an EOB demonstrates observably correct behavior—under a given event history EH_{end} resulting after the workflow execution—iff:

- it has reacted to all requests it is responsible for;
- it has reacted to all situations for which it defines workflow-specific rules and it is responsible for; and
- it has produced a reply or an exception for each request it has confirmed.

The first two items express the fact that an EOB has to react to all event occurrences if their types are defined in its event registration set and the EOB is designated as being responsible for them. Formally, the correctness of the observable behavior of an EOB b with respect to a complete event history (EH_{end}) can be expressed as follows:

Definition 6-11: (Correctness of the EOB behavior)

$$\exists eo_in \in EH_{end}: eo_in.type \in ESSET(b) \wedge b \in NS_{eo_in} \Rightarrow$$

$$(1) \exists t T(eo) < t < T(eo) + max_delay : contains(EDI, t, eo_in) = true \wedge \\ contains(EDI, T(eo) + max_delay, eo_in) = false$$

and also

$$(2) \forall r \in b.RULES: eo_in.type = r.on_etype \forall et \in r.GENTYPES \Rightarrow$$

$$\exists eo_out \in EH_{end}: eo_out.source = b.name \wedge eo_out.type = et \wedge \\ T(eo_out) > T(eo_in)$$

and also

$$\text{if } eo_in.type = request \Rightarrow$$

$$(3a) ((\exists eo' \in EH_{end}: eo'.type \in exceptions_of(service_of(eo_in.type)) \wedge eo'.origin = b)$$

or

$$(3b) (\exists eo', eo'' \in EH_{end}: \wedge eo'.type = confirmation_of(service_of(eo_in.type)) \wedge \\ eo''.type \in exceptions_of(service_of(eo_in.type)) \wedge eo'.origin = eo''.origin = b)$$

or

$$(3c) (\exists eo', eo'' \in EH_{end}: \wedge eo'.type = confirmation_of(service_of(eo_in.type)) \wedge \\ eo''.type \in replies_of(service_of(eo_in.type)) \wedge eo'.origin = eo''.origin = b)$$

Event occurrences are delivered to the *EDI* of EOB whose *ESSET* contains the event type by the *put()* operation, as soon as they are logged in the event history. The following description refers to the separate parts of the definition:

- (1) We require that delivered occurrences are consumed through the *consume* operation within a finite amount of time (*max_delay*). If the occurrence consumed is a request, a confirmation event is generated. If *max_delay* is exceeded, an exception is raised by *flush*.
- (2) We also require that once an event occurrence has been consumed, the rules which define reactions to that occurrence fire i.e., EOB rules express dependencies between consumed and produced event occurrences. Practically, this means that event occurrences for which the EOB was notified have been delivered and eventually triggered rules of the EOB.
- (3) In the event history resulting after a workflow has finished executing, for non-serviced or incompletely serviced requests there have to exist exception occurrences ((3a) and (3b)). On the other hand, for all requests that have been consumed by EOB, there must exist corresponding confirmation and reply occurrences (3c).

Given such a well-formed event history, we can argue that the EOB that participated in it demonstrated an observably correct behavior according to the REWORK specification or at the very least aborted (parts of) the workflow execution in a well-defined way.

6.5.3 Semantics of Workflow Execution

Based on the semantics of EOB operating on an event history, we can define the semantics of workflows over that history. The execution of workflows by EOB is reflected in the event history. Formally, the semantics of the execution of workflow instance w is the projection of the event history on those events that occurred within w : $EH \upharpoonright_{wid=w}$. Based on the mapping of specifications to EOB and services, a workflow execution is correct if each notified EOB reacts and generates adequate confirmations/replies or exceptions. Thus, if the event registration set of an EOB is adequately defined with respect to the intended workflow execution semantics, the workflow execution is correct if and only if each involved EOB behaves in an observably correct manner. Thus, the notion of correct behavior of single EOB expands to the correctness of a workflow execution: a workflow w is executed correctly under a given event history $EH \upharpoonright_{wid=w}$ if each EOB behaves observably correctly with respect to $EH \upharpoonright_{wid=w}$.

6.6 Summary

In this chapter we concluded our description of the REWORK metamodel. The actors participating in workflow execution are represented by various types of reactive composite components called event occurrence EOB. EOB generate and react to events in a REWORK system and transform these events to constructs understood by the represented actors. Thus, various kinds of real-world entities are described homogeneously by the constructs of the REWORK metamodel.

During workflow execution flexible task assignment policies can be defined with formulas expressed over organizational relationships between EOB, properties of the EOB, and of the events occurring in the system. From the perspective of the REWORK system, EOB are considered as black box components whose externally visible behavior has to conform to certain restrictions which define the semantics of correct workflow execution by an EOB. If these restrictions are met by each individual participating EOB, the correctness of the workflow execution by the entire REWORK system can be examined over the resulting event history.

7 Workflow Specification

In this chapter we turn our attention to the specification of processes with the constructs provided by the REWORK metamodel. As already mentioned, the metamodel provides the means to describe the architecture of a workflow system and the reactive behavior of the participating EOB. This behavior is specified by ECA-rules defined for the EOB.

In the previous chapter we defined the correctness of the observable behavior of an EOB with respect to a complete event history at a given time. This correctness concept takes into consideration the semantics of event composition and the notion of time in a distributed workflow system. However, it assumes that the mapping from the high-level workflow specification to the EOB behavior is correct, i.e., that we have specified appropriate EOB.

During the transformation of workflow specifications into REWORK systems, the REWORK build-time environment has to ensure that each actor is represented by an *appropriate* EOB. For this purpose, the described invariants offer a partial guarantee relating to structural system properties. Appropriateness is hereby defined in terms of static task assignment properties, rules, and replies/exceptions, and all of which have to conform to the role of the actor in the workflow specification.

The implementation of EOB on top of the REWORK run-time platform implementing the operational semantics of the REWORK metamodel then guarantees that they behave in an observably correct manner under resulting event histories—whereby, of course, we cannot ensure that the initial workflow specification itself is correct, i.e., conforms to the intended real-world meaning.

The underlying assumption in this chapter is that a high-level workflow specification language (e.g., graphical) complying to a workflow metamodel, such as the WfMC metamodel described in chapter 3, is used to define a workflow model. The resulting model is subsequently mapped to (elements of) a REWORK specification which when instantiated provides an operating workflow system capable of executing the defined workflows. In general, the definition of appropriate EOB for a given workflow specification depends on the intended operational semantics of the source workflow metamodel and on the appropriateness of the REWORK metamodel concepts to express the desired semantics. The approach is depicted in Figure 7-1.

In this thesis, we do not address the problem of mapping every conceivable workflow metamodel to the REWORK metamodel. Instead, we use in an exemplary fashion an established workflow metamodel as defined by the Workflow Management Coalition [WfMC, 1998] and describe a possible mapping to the REWORK metamodel. The WfMC metamodel has been chosen as it has various abstractions which are typical of a large class of workflow modeling languages. We also consider some general issues relevant to the mapping of workflow specifications to RE-

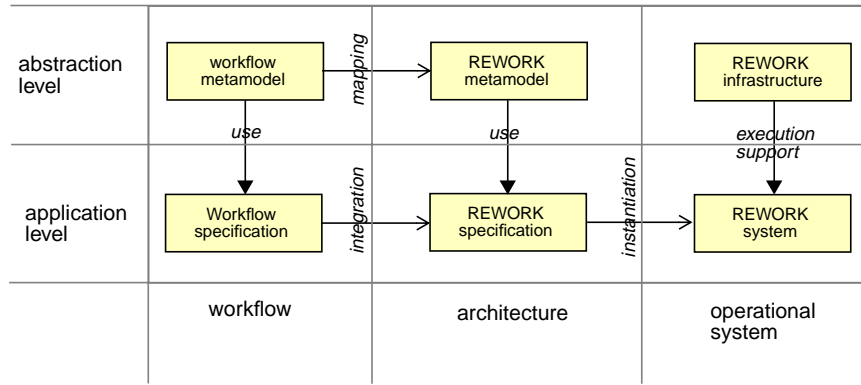


Figure 7-1: The role of workflow specification in the development of REWORK systems.

WORK specifications. Finally, we demonstrate how market-based task assignment can be implemented in a REWORK system through the definition of appropriate services and EOB behavior.

7.1 Integrating Workflow Specifications in REWORK Architectures

The main issue which we consider in this section is the correctness of the mapping of a workflow metamodel to the REWORK metamodel. The correctness property of the mapping refers to the *equivalence of the execution* of specifications abiding to the source workflow metamodel, as prescribed by its —explicitly or implicitly defined— operational semantics, with the corresponding execution of instantiated REWORK systems executing the specified workflows.

The ultimate goal of the mapping process between a source workflow metamodel and the REWORK metamodel is that it is completely automated. In that case, it can be implemented as an automated transformation function of the workflow composition environment. It is often however the case, that due to the semantic discrepancy between the workflow metamodel and the REWORK metamodel, different possible ways to integrate certain elements of the source workflow specification exist. In this case, the function of the composition environment is to provide adequate decision support to the workflow system developer.

Given a correct mapping from a source workflow metamodel to the REWORK metamodel, its semantics guarantee that workflows specified in the source metamodel are implemented correctly by the resulting REWORK systems. The proof of the correctness of the mapping ultimately depends on the semantics of the source metamodel. Thus, there can be no generic mapping algorithm and corresponding proof of the correctness of the mapping.

In general, the different kinds of dependencies between activities define part of the operational semantics of the workflow specification language. Under the assumption that the source workflow metamodel identifies atomic units of execution (i.e., atomic activities), and based on the externally visible state of these activities, different kinds of execution dependencies can be defined. Events and ECA-rules can be used to express dependencies between activities under the assumption that these dependencies are expressed over the state of these activities. State transitions correspond to events which are recognized in the system. ECA-rules are used to express state-transition dependencies. For our purposes, we define the correctness of the mapping of control flow to ECA-rules based on the notion of *equivalence of resulting event history to the intended execution history*.

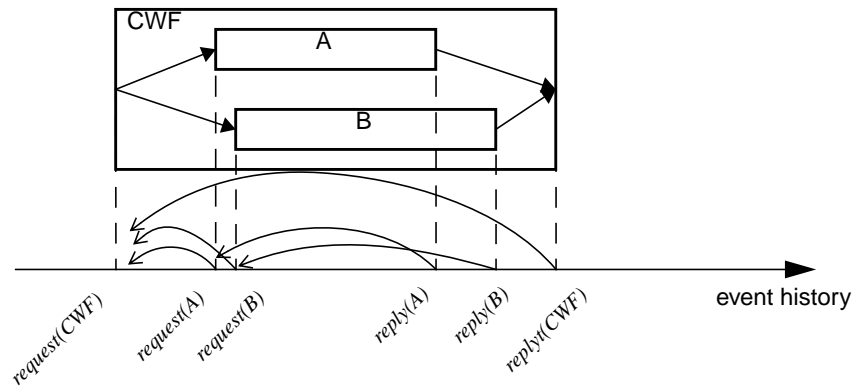


Figure 7-2: Nested activity execution

From a structural perspective it is important that the structure of the workflow specification is maintained during the transformation and can be inferred from the workflow execution. This means that primitive tasks are mapped to primitive execution units in the REWORK system. Furthermore, information concerning the eventual nesting of workflow execution must be maintained in the execution history resulting through the REWORK system.

As mentioned a primitive execution unit is a service whose initiation is denoted by a request event and its termination by one of the reply events defined for the service. The correctness requirement can be expressed over the event histories generated by the resulting REWORK system as follows:

- The specification of an atomic unit of execution *A* is mapped to a service *D*.
- When an atomic unit of execution from the source workflow specification is executed in a REWORK system, the event history must contain a request *request-D* and a subsequent reply event occurrence *reply-D* as defined by the service. These events must have identical origins and workflow identifiers.

Workflow specifications may allow the nesting of workflow activities (e.g., subflows in the WfMC metamodel). In order to be able to map the nesting of activity execution to the flat event history, the causal dependencies between events in the history are used. The approach is depicted in Figure 7-2, where nested activity execution corresponds to a sequence of event occurrences which hold references to their *causal antecedent event*. Through these references, it is possible to reach the first event in the history for a particular workflow execution, which denotes the initiation of the execution of the top-level workflow. The length of the path from a particular event occurrence in the history corresponds to the nesting level of the workflow specification. The causality relation between event occurrences can be maintained in the event history when appropriate references between event occurrences are created during the execution of ECA-rules generating new events, as well as from the correctness constraints posed on workflow execution by EOB. More specifically:

- request occurrences are causally dependent from some control flow rule which generates them (which may denote nested activity execution);
- service reply occurrences are causally dependent from the corresponding request occurrence; and
- service exception occurrences are causally dependent from the corresponding request occurrence.

7.2 Mapping the WfMC Process Metamodel to the REWORK Metamodel

In this section we describe an example mapping between a workflow metamodel and the REWORK metamodel. The source metamodel is the WfMC process metamodel [WfMC, 1998]. A WfMC-compliant workflow specification is called a *workflow process definition* (WPD). It describes the top-level processes as well as their relationships and attributes. Furthermore, it defines conventions for grouping WPD into related process data and the use of common definitions across different workflow models. It identifies the basic set of entities and attributes used in the exchange of workflow specifications. There are two WfMC-conformance issues which need to be clarified:

- The conforming system must provide a workflow specification language powerful enough to express at least the mandatory entities and their attributes. Thus the concepts described in the WfMC process metamodel must have corresponding concepts in the representation of the conforming system.
- The conforming system must provide an interface for the exchange of process models which is compatible with the WfMC Interface 1 specification.

The conformance guarantees that although a particular WFMS may use a custom internal representation, an import/export layer to the common *workflow specification language* (WSL) defined by the WfMC, allows the exchange of process specifications between different workflow definition clients and workflow execution engines. The principles of the process definition interchange as proposed by the WfMC are depicted in Figure 7-3.

The WfMC defines a “minimum metamodel” describing the core entities within a workflow metamodel. A workflow modeling environment should support the expression of these entities. It is thus important, that we establish the correspondence between these core entities and the elements of the REWORK metamodel.

In addition to the workflow specification perspective a conforming build time representation of the workflow metamodel is defined in the form of a “minimal state model”. This is necessary to define a number of basic principles and semantics of the execution environment. The correspondence between the minimal state model and the REWORK run-time model is also described in this section.

Each entity of the WfMC workflow metamodel is described by a set of mandatory and optional attributes. These attributes are divided into different groups:

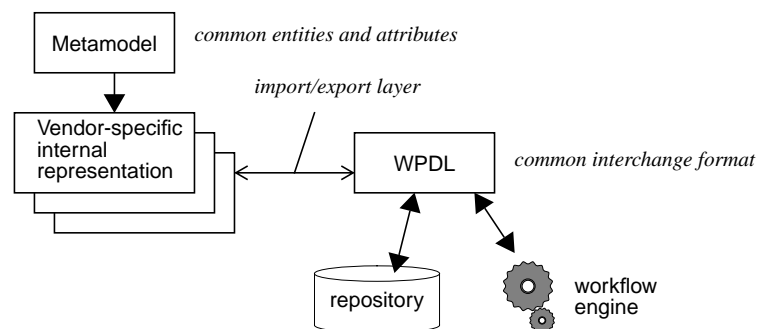


Figure 7-3: Process definition interchange according to the WfMC [WfMC, 1998].

- all entities have the attributes id, name, and description,
- attributes that characterize the respective metamodel entity,
- parameters, conditions, and values that refer to the workflow specification notation,
- attributes that reference other metamodel entities,
- documentation and icon attributes,
- attributes used for simulation and BPR, and
- extended attributes.

The metamodel allows the definition of further entities and attributes to create future conformance levels. For our considerations, the mapping of the mandatory specific entity characterization attributes is important. Other (optional) attributes are straightforward to implement as attributes of the corresponding REWORK metamodel abstractions. This will become clear as we describe the mapping. The WfMC process metamodel is depicted in Figure 7-4.

Various aspects of WFMS-specific extensions are not covered in the WfMC metamodel. These include the following:

- Extended attributes are used to define additional characteristics which need to be exchanged between systems. Run-time semantics associated with such attributes require bi-lateral agreement between exporter and importer.
- The local encoding and parameter passing scheme. The parameter passing scheme for workflow execution between different processing engines is defined in Interface 4 [WfMC, 1996b].
- Data type coercions for non-standard data types.
- Control flow internal to an activity. This refers to the specification of the sequence of work items and/or invoked applications which may be generated within an activity. The only provision made in the WfMC metamodel is an extended attribute which may offer two alternative invocation schemes —sequential or parallel. However, this approach is explicitly

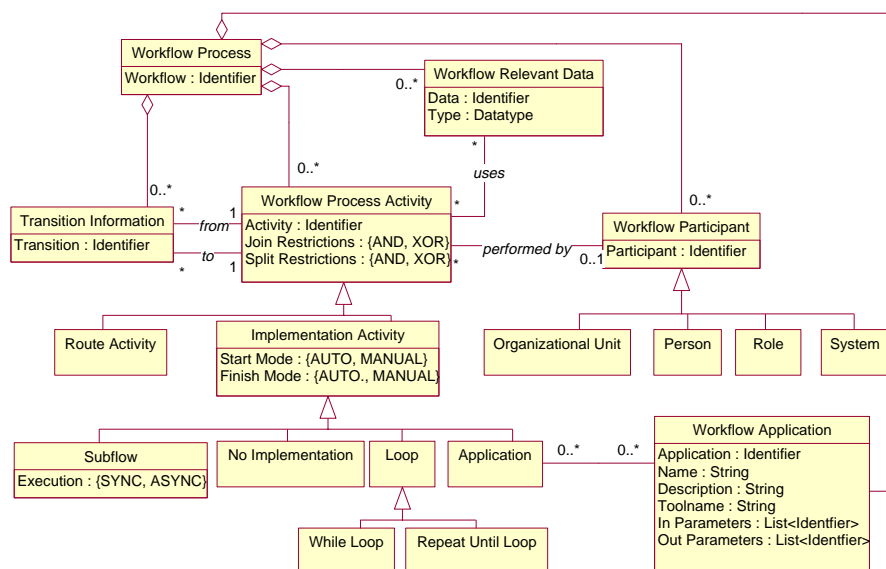


Figure 7-4: A class diagram [Booch et al., 1999] of the WfMC process metamodel.

discouraged and should be substituted by refining an activity into separate activities with their own flow control logic. This is also the approach followed by the REWORK meta-model.

7.2.1 Type System

The type system defined by the WfMC and used for the declaration of WfMC metamodel entity attributes is defined in the interface specification. Its mapping to the type system of the REWORK metamodel is straightforward.

The following basic data types are defined: boolean, float, integer, reference, string, and date. A reference is a string that holds references to external objects, e.g., file names, mail addresses, and URLs. Identifiers are sequences of letters, digits, underscores, and dots; they start with a letter or underscore. A performer finally is a type with value a declared workflow participant (see below). Complex data types can be records, arrays, enumerations, and lists with a variable number of elements.

7.2.2 Workflow Relevant Data

In a workflow specification *workflow relevant data* represent variables which store decision data used for control flow or reference data which are passed between activities and subprocesses. They do not include application data managed by invoked applications and which is not accessible to the WFMS and other applications. According to the scope of their definition, workflow relevant data can be accessed only by entities included in a process definition or may be used to define process parameters.

In REWORK systems, workflow relevant data are either passed between EOB executing a service through event parameters or are persistently stored in the state of EOB. The types of the data are expressed by the parameter type system of the REWORK metamodel (see chapter 5). Global workflow variables are stored as part of the state of EVE and can be directly accessed by ECA-rules defined for EVE.

7.2.3 Workflow Participants

A *workflow participant* in the WfMC metamodel is a potential workflow performer. A participant is defined in a workflow model as the recipient of a work item. Depending on the type of participant one or more actors may receive the work item. The different kinds of participants in the metamodel are represented by different types of EOB. More specifically:

- A *human* participant is represented by a user EOB in an REWORK specification.
- An *organizational unit* has a manager and various members. The unit is modeled by groups, users, and the *member_of* and *supervisor_of* organizational relationships in the REWORK system.
- A *role* is a set of skills or a function that a human has within an organization. In the REWORK metamodel a more general approach is provided. Groups of related skills are described by rule packages and state variables which can be associated with one or more EOB.
- A *system* participant in the WfMC metamodel corresponds to a caller or server EOB in the REWORK metamodel.

The constructs provided by the REWORK metamodel for representing the different types of workflow participants proposed in the WfMC metamodel are summarized in Table 7-1.

Table 7-1: Workflow participant representation

<i>WfMC metamodel</i>	<i>REWORK metamodel</i>
human	user EOB
organizational unit	groups, users, member_of and supervisor_of relationships
role	rule package
system	server, caller EOB

7.2.4 Workflow Process Definition

The *workflow process definition* specifies the elements that make up a workflow. It contains definitions or declarations of *activity* and optionally for *transition*, *application*, and *process-relevant data* entities. A workflow process may run as a subprocess invoked as an implementation of an activity of type Subflow; then parameters may be defined as attributes of the process. The only mandatory attribute is the process identifier.

A WPD is seen as a directed graph where nodes are workflow activities and edges are transitions that can be labelled by transition conditions. If such a condition is evaluated to true, the transition is enabled. If no transition condition is defined, the transition is always enabled. If multiple incoming or outgoing transitions are defined for a node, then additional control flow restrictions and condition evaluation semantics are defined.

A WPD corresponds to parts of an REWORK specification. REWORK specifications contain references to all defined specification artifacts. All REWORK artifacts have a unique identifier.

7.2.5 Transitions and Control Flow

In a WfMC conforming specification, transitions between activities and conditions which enable and disable them during workflow execution can be defined. A transition has a source and a target node. Two kinds of transitions are distinguished: “regular” and loop-connecting transitions (described below). Regular transitions can have conditions expressed on workflow relevant data; the conditions are boolean expressions. In REWORK specifications, these conditions are expressed as IF-conditions in ECA-rules.

A transition connecting a single source activity A to a single target activity B denotes a sequential ordering between A and B. In REWORK specifications it is expressed by a rule as depicted in Figure 7-5 which is defined for the EOB executing A. The event of the rule consists of the reply from the previous activity, the condition refers to the transition condition, and the action raises a request event for the following activity B. The condition can refer to the reply event parameters containing the results of the activity execution as well as to state variables of the EOB for which the rule is defined or to global state variables.

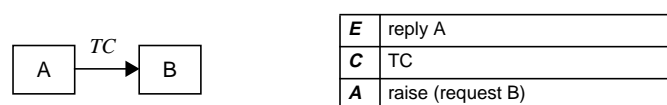


Figure 7-5: ECA-rule for activity sequence

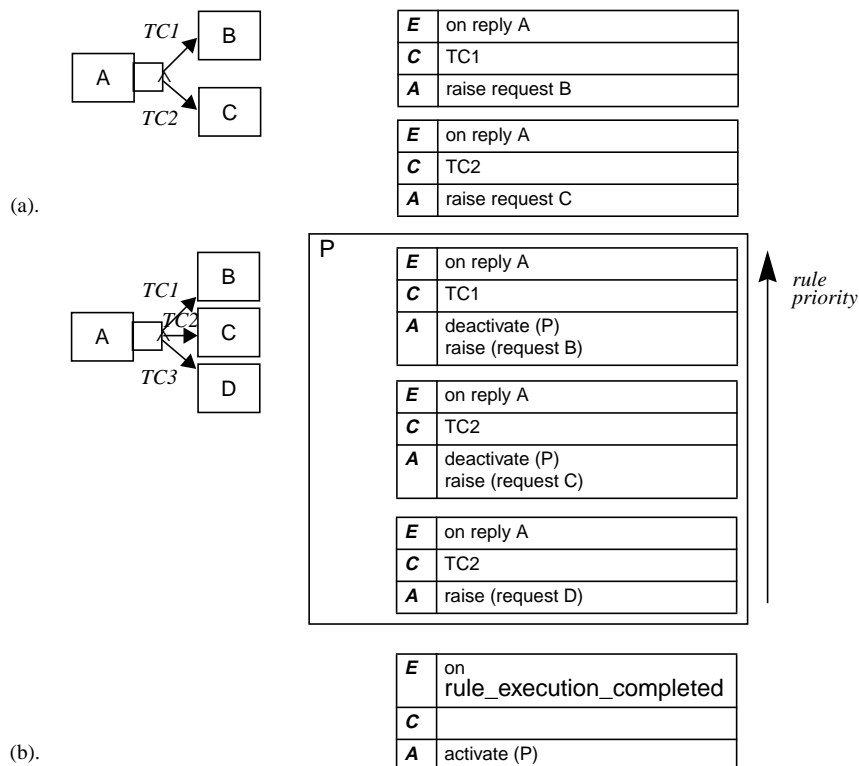


Figure 7-6: Mapping of (a). AND-split and (b). XOR-split in REWORK specifications.

A split element describes the desired control flow where multiple outgoing transitions for an activity exist. The mapping of AND- and XOR-split to the REWORK metamodel is depicted in Figure 7-6.

- An AND-split defines a number of possible concurrent threads represented by the outgoing transitions of an activity. The actual number of executed parallel threads is dependent on the conditions associated with each transition. An AND-split is mapped to rules in a rule package assigned to the EVE EOB. Each outgoing transition of an AND-split is mapped to an ECA-rule in the package whose event is the reply of service representing the source activity, the condition expresses the transition condition and the action raises a request event for the target service/activity.
- An XOR-split defines alternatively executed transitions. The decision as to which transition is selected depends on the individual transition conditions which are evaluated sequentially depending on their order in the list defined for the split element of the source activity. When an unconditional transition is evaluated or a transition with condition otherwise the list evaluation is ended. In an REWORK specification these semantics are mapped as follows to a rule package defined for the EVE EOB (see above). For each transition an ECA-rule whose event is the reply of service representing the source activity is defined. Each rule in the package has a higher relative priority with respect to the other rules in the package depending on the position in the list of the transition. The condition of each rule corresponds to the condition of the transition. The action of each rule deactivates the other rules in the package for the current rule execution cycle and raises the request for the corresponding target activity/service. Upon completion of the rule execution cycle, the deactivated package must be activated again. This must be transparent for the workflow specification and must be managed by the implementation of EVE. The limits of the rule execution cycle

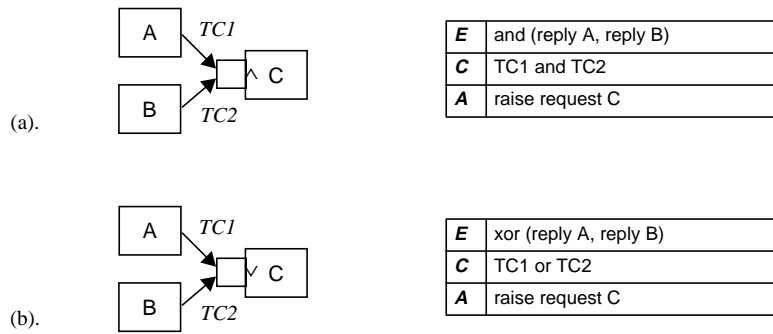


Figure 7-7: ECA-rules for (a). AND-join and (b). XOR-join

are denoted by two predefined special events in EVE, `start_rule_execution` and `rule_execution_completed`, which trigger rules executed first during the following execution cycle.

When multiple incoming transitions exist for an activity, a join element expresses the desired control flow:

- An AND-join is a rendezvous precondition for an activity and expresses the fact that the source node activities of all incoming transitions must be synchronized (i.e., have completed execution). An AND-join is expressed by an ECA-rule as follows. The event part is a conjunction of reply events for the transition source nodes. The conditions of the incoming transitions are conjoined in the condition of the rule. The action of the rule contains exactly one statement which raises a request event for the target node.
- An XOR-join indicates that any one of the source activities may have been completed in order for the target activity to execute¹. It is expressed by an ECA-rule whose event part is an inclusive disjunction of the source replies. The condition is a disjunction of the transition conditions and the action contains exactly one statement which raises a request event for the target node.

We note at this point that the WfMC defines different *conformance classes* with respect to the restrictions on the structure of the WPD graph. The semantics of split and join can only be accurately defined when considering these conformance classes. We mention here the most restrictive class which is the full-blocked. The adherence of a WPD to this conformance class requires that the following holds:

- For each join (or respectively split) there is exactly one corresponding split (or respectively join) of the same kind.
- In an AND-split no conditions are permitted.
- In an XOR-split an unconditional or otherwise transition is required if there is a transition with a condition (i.e., an undefined result of transition evaluation is not permitted).

The mapping of AND- and XOR-join to the REWORK metamodel is depicted in Figure 7-7. The requirements of the conformance class can be easily enforced by appropriate generation of ECA-rules in a REWORK specification.

¹ The term XOR-join is actually misleading as it does not correspond to the meaning of the term in logic. What is meant is that the join refers to an XOR-split.

In addition to the branching and joining operators defined in the WfMC process metamodel, additional kinds of logical operators may be defined in other workflow specification languages. These include for example, an inclusive-or split which specifies that a subset of the specified successors is chosen, and an inclusive-or join which specifies that a subset of the source activities have to be completed in order for the successor to start. The inclusive-or and inclusive split operators can be analogously mapped to (packages of) ECA-rules in REWORK specifications.

Furthermore, a different kind of dependency between activity execution can be defined instead of start/end dependency advocated in the WfMC metamodel. Thus a start/start dependency might be desired indicating that the target activity of a transition can start as soon as the source activity has started. In order to map this kind of dependency, the confirmation event of a service is used which indicates that an EOB has assumed the execution of the service. This for example, a dependency “B can start after A starts” is expressed by the rule R1 below:

R1	E	confirm A
	C	transition conditions
	A	raise (request B)

7.2.6 Workflow Activity Definition

Workflow activity definitions (WAD) define elementary activities that make up a workflow process definition. In general, WAD correspond to service types in the REWORK metamodel. However, not all of the WAD subtypes proposed by the WfMC need to be defined as services in an REWORK system. This becomes clear in the following sections.

Route Activities

The WfMC metamodel allows the use of special dummy activities called route activities for expressing cascading transition conditions such as,

```

if condition-1 then
  activity-1
else if condition-2 then
  activity-2
else
  activity-3

```

However when possible, the WfMC encourages the structure of such transition conditions as an XOR-split from the outgoing activity (see above). Furthermore, route activities are required in various cases such as:

- combination of XOR– and AND–split conditions on outgoing transitions from an activity
- combinations of XOR– and AND–join conditions on incoming transitions to an activity
- transitions involving conditional AND–joins of a subset of threads, with continuation of individual threads.

Route activities do not have a performer or an application assigned and their execution has no effect on workflow relevant data or application data.

Route activities are expressed as ECA-rules in REWORK specifications, which are assigned to the EOB EVE. The structure of these rules is explained below. It is however important to note, that the actions of these rules consist solely of raising new events. This reflects the fact that route activities have no effect on workflow relevant or application data.

Implementation Activities

The other category of activities identified by the WfMC metamodel includes so-called implementation or “regular” activities. These are expressed either a services or by ECA-rules in REWORK systems. Implementation activities have various implementation attributes. There are four types of implementation activities distinguished based on their implementation as described in the following paragraphs.

No implementation. Certain implementation activities have *no implementation* (!). These can be *manual*, in which case a human terminates the activity explicitly. Such manual activities are expressed by services in the REWORK metamodel for which the reply events are generated by an agenda EOB which is effectively operated by a human user. Activities with no implementation in the WfMC metamodel can also be automatic activities performed “implicitly” by the workflow engine. Such activities refer to pre- and post-processing of activities. In the REWORK metamodel, the concept of implicit activities is represented by synchronously requested services provided by some system component in conjunction with other services. ECA-rules are defined to couple these services as follows:

- If the implicit activity is a pre-processing for another activity, the event and action parts express the preconditions for starting the activity. The action part synchronously requests the pre-processing service and upon its completion executes the actual service. For example, assuming PREA is an implicit pre-processing activity for activity A then the following ECA-rules are defined for the EOB providing the service A:

R1	E	request A (and other preceding events)
	C	A start conditions
	A	raise (sync-request PREA)
R2	E	reply PREA
	C	–
	A	execute actions of A

- If the implicit activity is a post-processing activity POSTA for an activity A, POSTA is requested asynchronously by a rule defined for the EOB responsible for A which is triggered by the reply event(s) to A:

R1	E	A.reply
	C	–
	A	raise (request POSTA)

Application. Certain activities are implemented either by a WFMS component or some workflow system application. In the first case, the WfMC proposes that a library procedure identifier may be defined, while in the second case a tool identifier may be given. In the REWORK metamodel, a library procedure is a uni-extender EOB while a tool can be represented by a server or an extender EOB according to its implementation. A responsibility formula will be defined however, over the service implementing the activity which will return the eligible EOB. The implementation of application activities in REWORK systems is shown in Figure 7-8 (a).

Subflow. Another type of activity defined in the WfMC metamodel is that of a *subflow*, which means that the activity is a synchronously or asynchronously executed subprocess. An asynchronously executed subflow actually corresponds to an asynchronously requested service in the REWORK metamodel. A synchronously executed subflow is a hierarchical subprocess execution

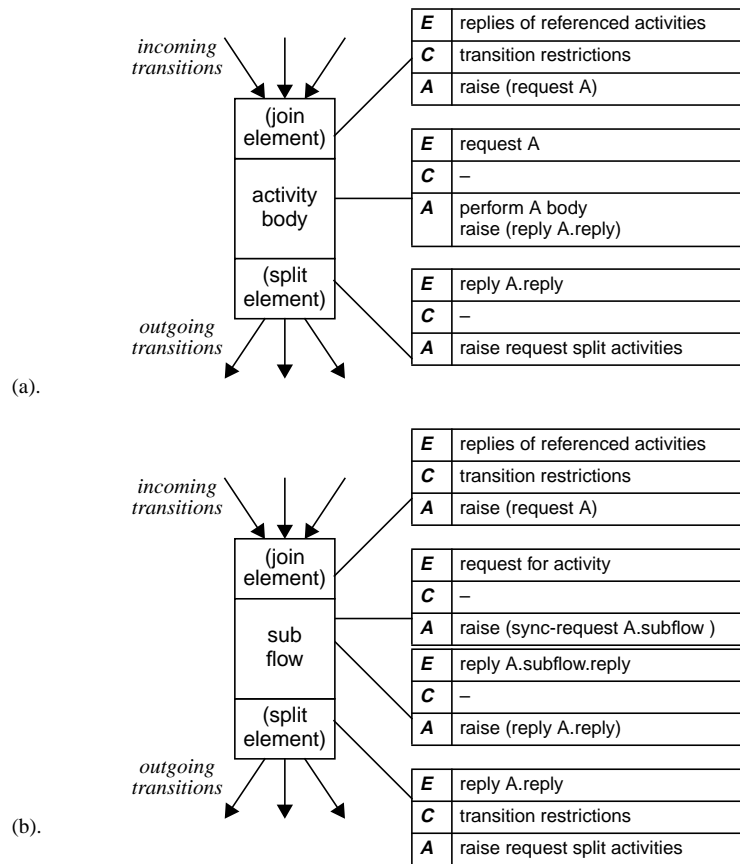


Figure 7-8: ECA-rules for (a). application and (b). subflow activities

and corresponds to a (sequence of) synchronous service request(s) in a REWORK specification. The mapping of subflow activities to REWORK specifications is shown in Figure 7-8 (b).

Loop. A *loop control* activity in the WfMC metamodel controls the execution of a *loop body*. It can either be a while loop or a repeat-until loop. Exactly one incoming and one outgoing transition exist between the loop control and the loop body. The loop control is represented by appropriate request and reply event types and ECA-rules. Without loss of generality, assume the while-loop control activity A with LB as the loop body:

```
while <condition>
do LB
```

For the mapping a new service is defined whose request signals the entering of the loop (request WHILE_LOOP_A), and the outcome of the loop evaluation (reply WHILE_LOOP_A.repeat and reply WHILE_LOOP_A.end). The ECA-rules derived for the mapping of a while-loop are assigned to the EOB capable of executing the first loop body activity. The resulting rules are depicted in Figure 7-9. A repeat-until loop is transformed analogously to a set of events and ECA-rules.

7.3 Market-Based Task Assignment

In this section we describe how the REWORK metamodel can provide support for the assignment of tasks to actors according to market-oriented criteria. According to this perspective, actors provide and consume computational resources, such as processor time or memory, during task execution. The EOB representing the client actor has to pay for a service execution in some notion of electronic currency. The EOB representing service providers are compensated for providing

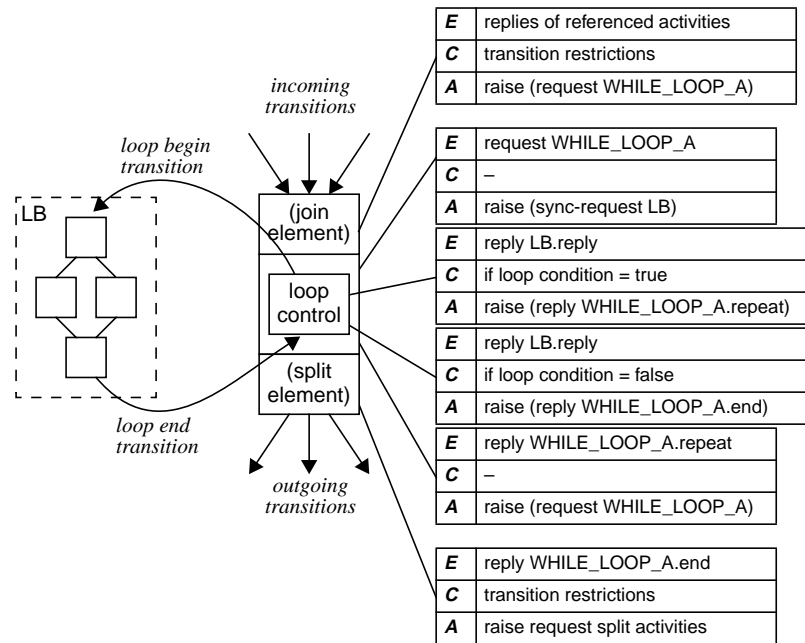


Figure 7-9: ECA-rules for while-loop activities

goods and services. Prices can thereby depend on factors such as how fast a service is provided, i.e., the faster a service is provided, the higher its price or the quality of the provided service, the provided quality, etc.

Market-based task assignment is based on certain assumptions about the information that is provided during workflow specification. In [Geppert & Tombros, 1997a] we describe the specification of market-based workflows with the TRAMs [Kradolfer & Geppert, 1997] language in detail. However, as is the case with the REWORK metamodel, the approach is not bound to a particular specification language. The sole requirement from the perspective of the REWORK metamodel is that the workflow specifications must provide information for the calculation of expected costs and execution times. Thus, for every atomic activity which must be assigned in a market-oriented manner, the expected execution cost and time must be specified.

Two underlying assumptions are made during the mapping of the specification to a REWORK specification:

- There are multiple EOB capable of providing marketed services. This assumption ensures that no monopoly exists. In fact, this is expected to be true in workflow systems that involve people performing intellectual tasks but also for common tasks which require physical resources, such as printing. If for every activity type exactly one capable EOB exists, a market mechanism is not meaningful.
- EOB do not agree on prices for service executions which implies that actors decide on prices independently of each other. This assumption is also basic in free markets, since its violation leads to cartels.

Depending on its provider, a service may have different cost and response times. In market-based task assignment, the interesting question is which service provider to choose in case there are multiple eligible ones. In order to support the selection of the optimal service provider, candidates have to publish information about their service execution attributes. This takes place during the actual workflow execution through an appropriate *bidding protocol*. This protocol assumes the

existence of an extender EOB called a *service market broker* (SMB) which is aware of the actual execution time and cost of activities, and implements the bidding protocol.

Market-based task assignment requires the specification of additional services which are used by the participants in the service market. A service `MarketBasedExecution` provided by the SMB with which the market-based assignment of service execution is initiated. The service has the following signature:

```
service MarketBasedExecution (string service_name, string request_parameters,
                               date reply_by) {
  replies:
    BestBidder ( string EOB_name, integer cost, integer duration, date valid_until);
  exceptions:
    NoBidder ( );
}; // MarketBasedExecution
```

The SMB which provides this service must implement a bidding protocol through which the optimal bid is calculated. This is based on a further service `CalculateBid` which must be provided by all EOB which participate in the market. We note here, that the broadcasting of asynchronous request events provides direct support for the implementation of open interactions (such as bidding) between EOB. In the simplest case of supporting bid calculation only based on time and cost parameters, the `CalculateBid` service is defined as follows:

```
service CalculateBid (string serv_name, string request_parameters, string client_name) {
  replies:
    BidReply ( integer cost, integer duration, date valid_until );
}; // CalculateBid
```

The SMB requests the `CalculateBid` service and sets a timer for the end of the bidding period. The timer is set by use of a further service `Notify` provided by an extender EOB which provides time and date services. All replies to the service are collected until this bidding period expires, i.e., a

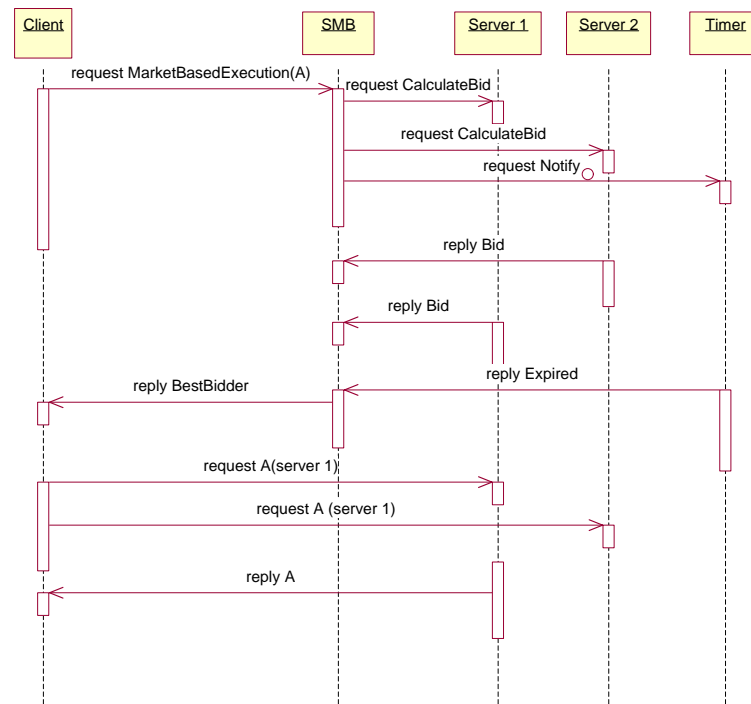


Figure 7-10: Sequence diagram [Booch et al., 1999] of the bidding protocol for market-based task assignment. Servers 1 and 2 are capable of providing service A; server 1 is contracted for service execution.

bid collection rule triggered by an iteration event type `iter(CalculateBidBidReply, Notify.Expired)` is defined for the SMB. The best bid is calculated then and the client of the service which is to be executed in a market-based manner and all bidders are informed of the winning bid and its provider. At this point a service provision *contract* has been made between the client and the server. The entire bidding process is depicted in the sequence diagram [Booch et al., 1999] of Figure 7-10. More complex bidding and contracting protocols are conceivable (e.g., two-phase contracting), but are not elaborated here as they can be implemented with similar mechanisms as the described protocol.

The participation of EOB in market-based task assignment is enabled by extensions to their state and their behavior. The state of each service provider has an account attribute collecting the amount of electronic currency it has earned by service executions. In order to participate in the bidding process, each server must be capable to provide the `CalculateBid` service. Furthermore, the condition of the rule(s) defining their reaction to the marketed service must filter if they have signed a contract (as explained in the next paragraph). Finally, the server EOB must be able to persistently store information about contracts it has to honour with specific clients.

The client EOB must use the `MarketBasedExecution` service each time it desires a market-based execution of a particular service. Once the winning bidder is determined, it requests the service with an additional parameter which denotes who the contracted server is¹. The server is then responsible for providing a reply. Thus an appropriate suitability formula filtering the `EOB_name`-parameter has to be defined.

7.4 Summary

In this chapter we described the mapping of workflow specifications to REWORK specifications. The mapping of the WfMC process metamodel was demonstrated in some detail. In general, the correctness of a mapping depends on the intended operational semantics of the source metamodel. The REWORK metamodel provides the concept of causal relations between event occurrences which can be used to ensure that the causal relationships between execution elements of the source specification are maintained. This was demonstrated for the example of nested activity execution.

The power of the REWORK metamodel was demonstrated by the specification of market-based task assignment. By the definition of an extender EOB, appropriate services, and market participant behavior it is possible to support a bidding process for the efficient execution of services. Analogously, other extensions to the basic functionality of the WFMS infrastructure can be made through definition of new services and providing EOB. A monitor component for workflow execution has been implemented as part of this thesis demonstrating the feasibility of the approach.

¹ This means that the definition of services which are to be executed in a market-based manner must be appropriately extended. Their request parameters include the contracted server.

Part III: Workflow System Composition

In the third and last part of this thesis we turn our attention to the issues concerning the implementation of REWORK systems. From the perspective of the workflow application development process proposed in chapter 1, part III considers the build-time infrastructure required for the composition of REWORK systems and the run-time infrastructure (i.e., the implementation mechanism) required for the operation of composed REWORK systems.

In chapter 8 we describe the overall organization of a workflow composition and execution environment. Furthermore, we describe the repository-based REWORK system composition environment.

In chapter 9, we describe the run-time infrastructure required for distributed event-based workflow execution. We especially consider issues pertaining to distributed composite event detection and discuss some implementation alternatives.

Finally, in chapter 10 we describe an entire workflow system lifecycle based on the REWORK framework. We discuss the individual phases of REWORK system composition, the various mechanisms applied in each phase, as well as the input and output artifacts of each phase.

formal exposition of the algorithms and data structures, in REWORK service definitions, behavior and state of the EOB types. The *abstraction realization* corresponds to the source code produced when EOB types are instantiated. The general problem for the implementors of software schemas is to find suitable abstractions for representing the schemas. In this thesis, the representation chosen is the REWORK metamodel and the hereby defined formal semantics for event-based workflow execution.

The composition of REWORK systems requires the support for the storage, location, and selection of appropriate artifacts. This functionality is provided by a build-time repository described in the next section.

8.2 A Build-Time Repository for the REWORK Metamodel

In this section we describe the requirements and functionality of a build-time repository for REWORK system composition. Furthermore an implementation of such a repository on top of the object management system Shore is described.

8.2.1 Requirements

The issues concerning the development of effective and usable repositories for software artifacts represent an active field of research (e.g., [Constantopoulos & Dörr, 1995, Henninger, 1997]). In our work we have attempted to utilize the experience and results gained from this research within the context of workflow system composition. The specific requirements that have to be met by a build-time repository for REWORK systems are briefly described in the next paragraphs. For general requirements on design database systems we refer to the elaborations in chapter 2 and to the appropriate literature.

- *Completeness*: The repository structure must allow the storage of complete REWORK systems. Thus, all the different types of artifacts definable by the REWORK metamodel must be represented in the repository. Furthermore, the classification scheme underlying the architectural analysis of workflow systems (see chapter 4) must be supported.
- *Invariants*: The invariants defined in the metamodel must be expressed by corresponding artifact associations.
- *Artifact state*: The consistency maintenance of artifact definitions in the repository must be supported. This means that different artifact states must be supported: in-development and in-production. Objects whose state is in-production must be consistent with respect to other objects and completely defined. Furthermore, different versions of the same artifact must be supported. Artifact version management issues have not been considered within the context of this thesis.
- *Indexing*: The indexing structures of the repository plays an important role for the effective retrieval of artifacts. Workflow system composers must be able to efficiently search for components which have a specific trait. Consequently, appropriate indexing structures must be defined.
- *Queries*: While it is generally accepted that facet-oriented classification schemes —such as the connotation-based classification— make it simpler to combine the terms that describe components, it may become hard to find the right combination of terms that accurately describe the information need and the result may be a large set of components [Hen-

ninger, 1997]. Thus, textual queries on the attributes of components defined in the REWORK metamodel must be supported. Consequently at least rudimentary query processing capabilities must be provided by the repository.

8.2.2 Implementation of a Build-Time Repository with Shore

A prototype of a build-time repository for the storage of REWORK artifacts was built around the object management system Shore [Carey et al., 1994]. *Shore* (Scalable heterogeneous object repository) is being developed at the University of Wisconsin and at the present time is available in release 1.1.1. Shore represents a development of an earlier system (EXODUS) and addresses specific limitations of that system, including lack of storage of type information, its limited scalability, the lack of support for access control, and limitations of its application programming interface.

Shore is a collection of distributed cooperating data servers (see Figure 2-4) with a shared UNIX-like name-space. As in UNIX, named objects can be directories, symbolic links or individual objects. Objects can be accessed through a globally unique object identifier. The Shore type system is language neutral and supports applications in any programming language for which an appropriate binding exists. As already mentioned, Shore executes as a group of communicating processes called Shore servers. They consist of trusted code and manipulate fixed length pages allocated from disk volumes, each of which is managed by a single server. Applications—which are not trusted—can only modify objects and are not authorized to access meta data such as indexes or directory structures. The servers function as page-cache managers for pages from local as well as remote volumes. They additionally provide concurrency control and recovery by obtaining and caching locks on behalf of their client applications. Thus from an application viewpoint the location of the data is transparent and thus is accessed in an identical manner. Furthermore the caching of remote pages at a local server has the potential of reducing the performance penalty of accessing remote data.

Shore provides a tree structured name space in which all named persistent objects are reachable from a distinguished root directory. The users of Shore are thus provided with a framework to register individual persistent objects and the roots of large persistent data structures. The framework provides basic concepts such as directories, symbolic links and “files” as well as special concepts such as cross-references, pools of unnamed objects, modules, and type objects.

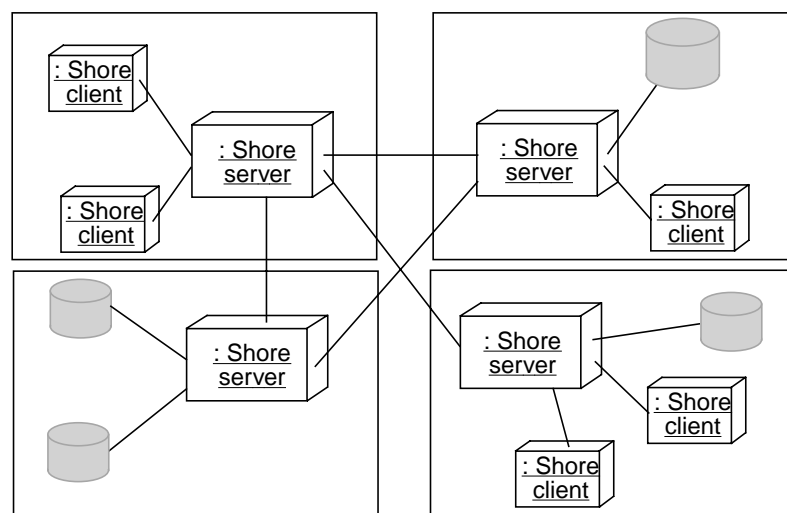


Figure 8-2: An example model of the Shore system organization.

Most of these facilities are used in the implementation of the build-time repository. The provided object model consists of objects and values. All persistent data are objects with unique identities. Structurally, objects are containers for simple or structured typed values which may include references to typed object identifiers of other objects. Each object has a set of methods to access and manipulate its contents. The internal structure and methods available for an object are described in its interface type. Every Shore object is tagged with a reference to a type object which contains this information.

8.2.3 The Build-Time Repository Schema

The build-time repository stores all REWORK system artifacts in appropriate objects. Additionally administrative information has to be stored. The build-time repository schema consists of about 60 interface types and is divided among various SDL modules.

As described in chapter 6, an REWORK system consists of a set of workflow system components, a set of event types, a set of suitability formulas, a set of dispatch formulas, and a set of organizational relationships. The repository stores the complete representation of an REWORK system. The REWORK metamodel constructs are modeled in the build-time repository by Shore interface types. REWORK system artifacts are stored as named objects of these types.

The Shore Data Language

Shore interface types are described with the Shore Data Language (SDL), which is based on the Object Database Management Group (ODMG) ODL language specification [Cattell et al., 1997] and supports the language-independent description of these data types. As mentioned previously, interface types can have attributes, methods, and relationships. The attributes can be one of the built-in primitive types (e.g. integer, character) or can be constructed through type constructors such as enumerations, arrays, or references. Several bulk data types are provided which enable an object to contain collections such as sets, lists or sequences of references to other objects (see Figure 8-3).

The provided SDL type compiler compiles the interface definitions into type objects stored in the database and target language stubs. Currently, the only binding supported by Shore is to C++.

```
interface REWORK_EventType {
protected:
    attribute REWORK_Types type;
public:
    attribute string name;
    attribute sequence<ref<REWORK_ECARule>> rules;
    attribute ref<REWORK_Timestamp> sync;

    relationship ref<REWORK_EventDetector> detector inverse etype;
    relationship set< ref<REWORK_Site>> send_to inverse events;
    relationship ref<REWORK_Interval> interval inverse restricts;

    void initialize(in lref<char> nname, in lref<char> site);
    void finalize();
    void display(inout ostream os) const;
    long assignIntv(in ref<REWORK_Interval> intvl);
    void setDetector(in ref<REWORK_EventDetector> det);
    long addRule(in ref<REWORK_ECARule> erule);
}
```

Figure 8-3: The SDL specification of the supertype of REWORK event types.

A Java binding is planned however. The type compiler generates a set of class declarations (a C++ header file, Figure 8-4) and special-purpose functions (C++ source file). Collections for example are represented in C++ using predefined template classes (parameterized types). For example, the class `Ref<REWORK_Timestamp>` encapsulates an OID; The overloading features of C++ make it behave like a pointer read-only instance of `REWORK_Timestamp`. The class `SetInv<REWORK_Site, ...>` encapsulates a data structure containing a set of OID pairs and it provides member functions that enable its contents to be accessed, e.g., the function `members` that returns an iterator.

The combination of the SDL compiler and a run-time library allows programmers to write applications that manipulate objects through type-safe object references. The library takes care of fetching objects on demand to an LRU client-level object cache, flushing changes to the server on transaction commit, and swizzling and unswizzling references as necessary. Given the generated header file, the Shore application programmer can implement the operations associated with the provided interfaces. The access to data members is made safe e.g. by generating read-only pointers or by flagging member functions that do not update the contents of an object as `const` in their SDL definition. In order to modify an object the C++ application must call the special generated member function `update` which returns a read/write reference.

Storing REWORK Specifications

As described in chapter 6, a REWORK specification consists of a set of EOB specifications, a set of event types, a set of suitability formulas, a set of dispatch formulas, and a set of organizational relationships. These REWORK artifacts are mapped to the build-time repository as follows:

- For each participation attribute defined in chapter 4 an aggregate object of type `ParticipationAttr` exists in the build-time repository. The object references `Terms` —which can be composite objects in themselves— defined for the attribute values. A term contains a collection of references to objects of the type `Actor`. Actor objects may be optionally associated with an EOB object. The structure is exemplified for two traits in Figure 8-5. It is important to note that each defined Actor must be associated with every individual `ParticipationAttr` object in the repository.

```
class REWORK_EventType : public sdlObj {
    COMMON_FCT_DECLS(EventType)
protected:
    enum REWORK_Types type ;
public:
    sdl_string name ;
    Sequence< Ref< REWORK_ECARule > > rules;
    Ref< Timestamp > sync ;
    Ref< Site > home ;
    RefInv<REWORK_EventDetector, REWORK_EventType,36> detector;
    SetInv<REWORK_Site, REWORK_CEventType,40> send_to;
    RefInv<REWORK_Interval, REWORK_EventType,52> interval;

    virtual void initialize ( LREF(char) newname , LREF(char) sitename );
    virtual void finalize ();
    virtual void display ( class ostream &os ) const ;
    virtual long assignIntv ( Ref<REWORK_Interval> intvl );
    virtual void setDetector ( Ref<REWORK_EventDetector> det );
```

Figure 8-4: The C++ header generated for Figure 8-3.

- Each EOB, is represented by an aggregate EOB object. The structure and implementation of the EOB object depends on the EOB type. An appropriate interface type is defined for each EOB type. Components of an EOB are instances of defined STATEMAN, ACTMAN, and ECAMAN objects. These contain collections of references to StateVariableType objects, OperationType objects, and ECARule objects. Implementations of ACTMAN are stored as symbolic links to object files.
- Organizational relationship types are represented as OrgRelType interface types. Their attributes are strings of the permissible participating source and target EOB types. The rudimentary meta-object facility provided by Shore is used to determine if a permissible EOB type participates in OrgRel which are objects containing references to the source and target EOB objects. More specifically, the Shore TYPE_OBJECT(T) macro yields a reference to the compiled-in C++ type object for T allowing the casting between types and as a side-effect the checking of object types. Organizational units are represented by OrganizationalUnit objects.
- Services are represented by Service interface types. Furthermore, for each service definition a corresponding Request and Confirmation event type object is created. For each Reply and Exception event type a corresponding reply and exception event type object is created. Confirmation, reply, and exception event type objects have bi-directional references to the respective request event type object. Event type parameter objects are created for each event parameter based on corresponding interface types.
- An interface type ECARule for ECA-rules is defined. It contains references to the event type objects in its event clause. Composite event types implicitly defined in the rule event clause are represented by objects of the appropriate interface type. These are created when the rule object is stored in the repository. An ECARule also contains references to conditions. Conditions are sequences of triples of object references of the form ($\{AND|OR|NOT\}$, *name*, *type*, *logical_operator*) where *name* and *type* are names of corresponding objects defined in the repository, the *logical_operator* must be defined for the type, and $\{AND|OR|NOT\}$ determine the relationship to the previous element (the first element in the sequence has an AND value defined). The precedence order of the expression determines the order of sequence elements. Actions are sequences of *operation_name* strings where *operation_name* is a name of an operation type defined for the EOB, or a raise operation for an event type. The corresponding object references can be dynamically retrieved through predefined queries. Additionally, actions of extender rules can contain symbolic links to source and object code files.

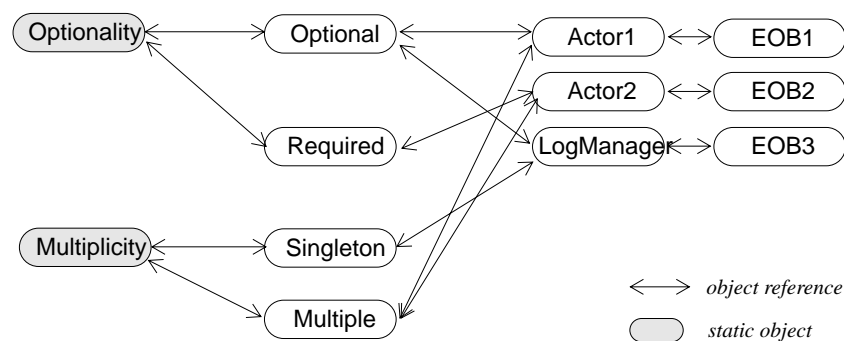


Figure 8-5: Example of participation traits and processing entities in the buildtime

```
SH_DO(Shore::init(argc, argv, getenv("REPOSITORY_NAME")));

SH_BEGIN_TRANSACTION(rc);
if (rc)
    rc.fatal();

BuildtimeRepository::init(WARM, true); // read static object from the database

// call build-time interface operations

if(errcount == 0)
    SH_DO(SH_COMMIT_TRANSACTION);
else
```

Figure 8-6: The generic coding structure of buildtime-repository applications.

- Suitability and dispatch formulas are represented by composite objects which are again references to the participating elements.

In general, during the creation of objects in the build-time repository, some objects are in the in-development state (e.g., because they reference names of as yet undefined objects). This is allowed during the composition phase of a new REWORK system. However, when the REWORK system is instantiated, every participating object must have the in-production state. This is ensured by the instantiating transactions which create objects in the run-time system and ensure that the REWORK system invariants are respected.

The creation and storage of REWORK artifact objects in the build-time repository objects is performed by client applications of the Shore database. These applications include interactive editors and a compiler. In the context of this thesis simple interactive textual interface tools were built to support the creation of repository objects. Alternatively the compiler tool `bscc` uses textual input to populate the build-time repository. The approach chosen is extensible, as all build-time tools use exclusively a build-time interface to access the repository.

The build-time interface is based on a static repository object stored in a Shore database. The repository object interface consists of a set of C++ operations which can be used to create, retrieve, and manipulate repository objects. These operations perform all necessary consistency checking when creating a new repository object. The interface is available through a library which can be linked with development tools. In the current implementation the manipulation of the build-time repository occurs in a single design transaction. While the splitting of the repository manipulation operations to multiple transactions is possible, it has not been examined in the context of this thesis. Consequently, currently each client application has the general structure depicted in Figure 8-6.

8.3 Summary

In this chapter we described the implementation of a build-time repository for REWORK system composition. The repository is built on top of the object-oriented DBMS Shore. The repository schema allows the complete definition of REWORK systems and consequently allows the control of the structural completeness of a developed REWORK system.

9 The REWORK Run-Time Architecture

In this chapter we turn our attention to the implementation of REWORK systems. We describe the requirements for a run-time environment for REWORK systems, the run-time architecture of REWORK systems, and various aspects of a prototypical proof-of-concept implementation realized at the University of Zurich. A substantial part of the work described in this chapter has been done in collaboration with other researchers and is documented to some extent in [Geppert & Tombros, 1997, Geppert & Tombros, 1997a, Geppert & Tombros, 1998].

From the perspective of REWORK systems, a lightweight run-time communication and coordination infrastructure is required to allow their execution. The run-time environment must provide support for the implementation of EOB and for the interaction between them. Furthermore, it is desirable that the mapping between the REWORK system build-time representation and the run-time environment is as straightforward as possible. To support this, the developed run-time environment consists of an asynchronous messaging layer, an event notification and composition system, and a run-time repository for REWORK systems. As the primary goal of the system is event-based workflow execution, it is appropriately called EVE for *event engine* [Geppert & Tombros, 1998]. The following functionality is provided by the EVE run-time infrastructure:

- *Distribution.* EVE provides the middleware layer for workflow execution by actors. Actors typically are distributed over various hosts (or sites) connected by the Internet or an intranet. This requires distributed event detection and messaging facilities. Furthermore, location transparency is provided to participating EOB.
- *Failure resilience.* The connectivity of actors over the network may dynamically vary over time —as is the case when a site host or a sub-net crashes. This requires some failure resilience which is provided by persistently stored network messages in EVE servers and fail-safe message transmission with exactly-once semantics.
- *Run-time repository.* In order to maintain the necessary meta information to operate a REWORK system, global naming, persistence, and retrieval services are needed. These are implemented by the *EVE databases* which persistently store and manage information about EOB, event types, and other information pertaining to the run-time REWORK system architecture.
- *Workflow execution.* In order to execute workflows (in an event-driven style), event detection, event notification, and task assignment services are needed.
- *Event history management.* Various workflow applications require logging, monitoring, and analysis services. These services rely to a large extent on the persistent event histories maintained by EVE.

The implementation of this functionality in the EVE prototype is discussed in the following sections. The overall architecture of EVE run-time environment is depicted in Figure 9-1. An EVE system consists of a collection of EVE sites which are host machines on which a Shore database server resides (see chapter 8). Each of these sites consists of a collection of persistent and non-

persistent objects: the non-persistent objects, called the *EVE clients*, live outside the Shore database, while their persistent counterparts are the EVE server objects. The server objects of a particular site as well as the event detection subsystem compose an *EVE server*.

The EVE clients use a common *message service subsystem (MSS)* to communicate with their server objects. The MSS ensures the reliable communication with the site-local clients and the other servers in the system. Furthermore, EVE clients communicate with external applications through application specific interfaces. Each site has a server object in every other site in the system so that server-to-server communication is also performed through the MSS as is client-to-server communication.

9.1 EOB Implementation

EOB are implemented as Shore applications. More specifically, the ACTMAN subcomponent of each EOB is implemented as a Shore client process, while the ECAMAN and STATEMAN subcomponents are implemented by persistent composite objects in Shore. The run-time structure of the ECAMAN and STATEMAN subcomponents depend on the definition of the EOB in the build-time repository.

The ECAMAN consists of rule execution subcomponent which executes rules stored in the EVE database, and an event generation subcomponent which generates workflow system events in response to rule execution. Rules are composite objects which reference compiled C++ code for condition evaluation and rule actions. Conditions are expressed by queries over the local EVE database and actions send messages to Shore objects. The ACTMAN is implemented as a persistent composite object which contains references to a further (composite) object for each state variable defined for the EOB.

The EDI and EPI of the EOB are container objects which consist of collections of references to event occurrence objects stored in the event history database. Furthermore, the state of the EOB is again stored in the EVE database as a collection of persistent objects referenced by the rule objects.

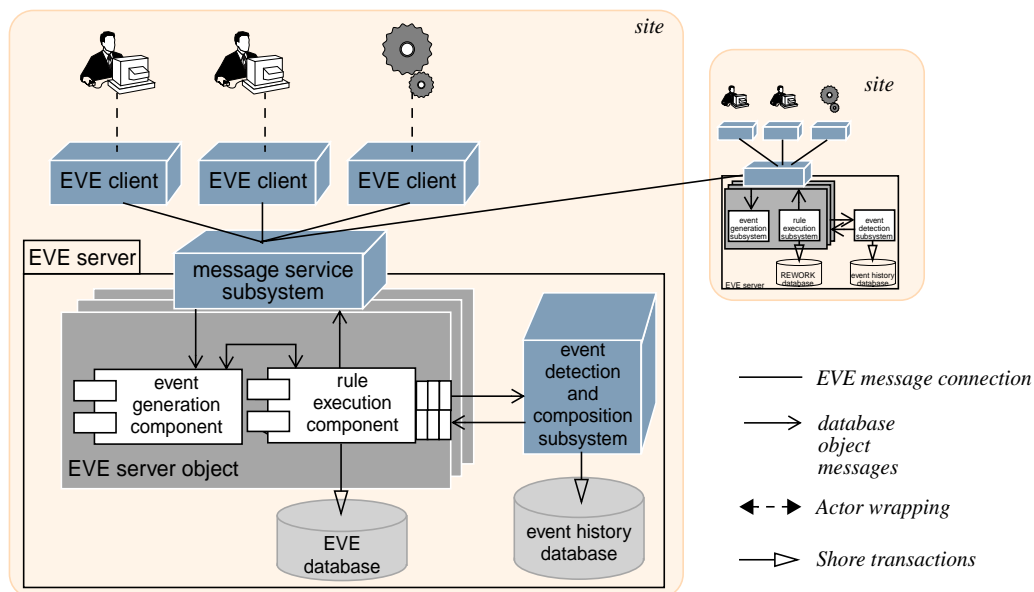


Figure 9-1: The implementation architecture of EVE.

The client part of an EOB is a process which interacts with external applications, i.e., ACT-MAN for EOB of the types external, user, and server. It is implemented either as C++ or a Java application. It consists of an application-dependent part realizing the implementation traits for automation, server type, and interface (see chapter 4) of the actor, and an application-independent part, called the *EVE adapter* which provides messaging services. The EVE adapter is a C++ library or Java package linked with the rest of the client code.

We note that for each EVE server in the system, an appropriate extender EOB exists in every other server. Thus, a complete topology of the system servers and their respective EOB clients is available to every server. The EOB representing remote server are used to forward interesting event occurrences. Consequently each time an event that has remote listener EOB, the extender of the corresponding EOB server receives an appropriate message and forwards it to the remote site. In that case the MSS is used to directly contact that remote server.

9.2 Messaging

The distributed messaging subsystem of EVE is built as an extension of the Adaptive Communication Environment (ACE) [Schmidt, 1999]. ACE is a freely available open source object-oriented framework that implements various core design patterns for concurrent communication software. It provides a library of reusable C++ wrappers and framework components that perform common communication software tasks across a range of OS platforms. Its purpose is to provide an integrated set of components that help developers navigate between the “Scylla and Charybdis” limitations of (1) low-level native operating system API, which are inflexible and non-portable, and (2) higher-level distributed object computing middleware, which are often inefficient and unreliable. As such it provides the ideal portable communication layer required for the implementation of EVE. Particularly interesting for our purposes was the existence of a Java version of ACE (JACE) which is interoperable with the C++ version. The overall organization of the framework is depicted in Figure 9-2.

The communication software tasks provided by ACE which are used in EVE, include event de-multiplexing, event handler dispatching, and message routing. The *acceptor*, *reactor* and *connector* components were extended for our purposes, allowing to encapsulate the network interfaces of the underlying operating system from the perspective of EVE. Furthermore, a message

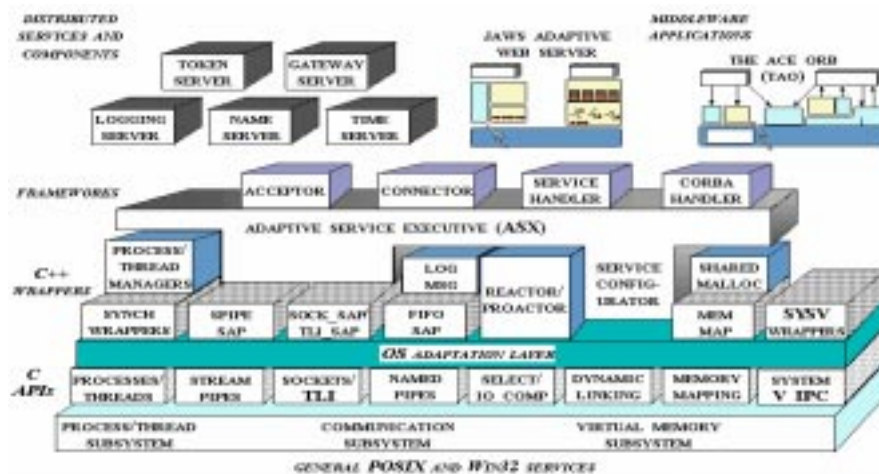


Figure 9-2: The layered architecture of components in the ACE framework [Schmidt, 1999].

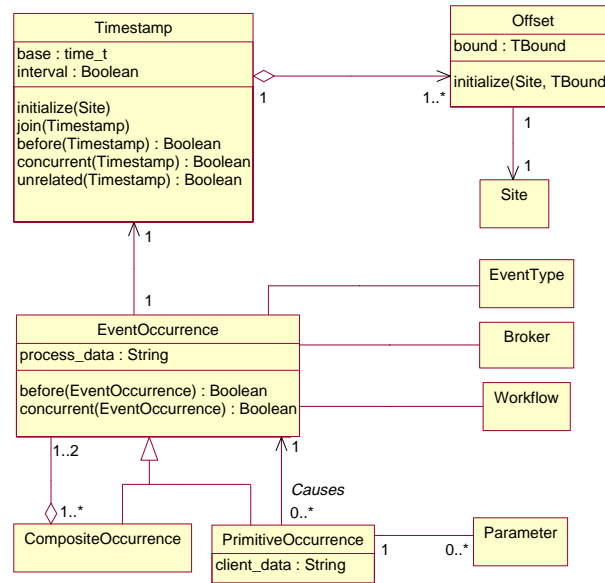


Figure 9-3: Class diagram [Booch et al., 1999] of event occurrence and related types in the run-time repository of EVE.

class library was developed to express the various types of network messages exchanged between distributed EVE system components. The exchanged messages are of the following types:

- operation call messages, determining the operation which must be executed by the EVE client;
- operation result messages, returning the result of the operation to the server; and
- event messages sent to other servers in the system (see below).

A CORBA-compliant implementation of the messaging subsystem was not considered for the prototype, but is possible through the use of the real-time ACE ORB [Schmidt et al., 1998].

9.3 Distributed Event Detection

In this subsection we consider the realization of event detection in EVE emphasizing the aspects of distributed composite event detection.

9.3.1 Event Occurrences

Event occurrences are the *actual happenings* of interest at some point in time and are considered as instances of event types. Event occurrence objects are stored in the event history database in SHORE with the following attributes:

- a reference to the event type object for the occurrence,
- a reference to the occurrence site object,
- a unique site-specific occurrence identifier,
- a reference to a new timestamp object defining when the event occurred,

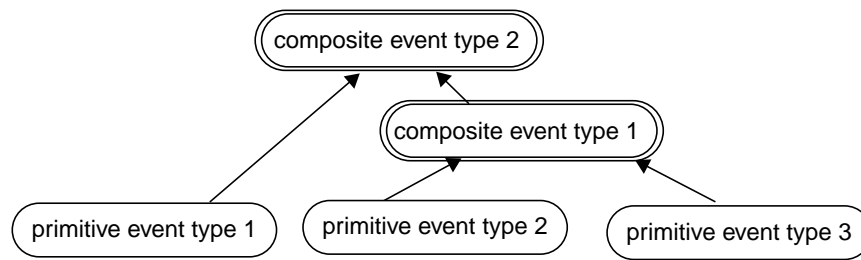


Figure 9-4: Event detector graph. Event detector objects are generated for each defined event type.

- a reference to the event occurrence on which the event causally depends (its antecedent event occurrence),
- a reference to the workflow instance identifier within which the event has occurred,
- a reference to the EOB which generated the event,
- a list of component occurrences in case of composite events, and
- a list of references to typed event parameter objects corresponding to the data parameters of the event.

Timestamp objects have a complex structure themselves. They consist of a global *base* time and an offset for each site participating in the timestamp. Each offset indicates if the site time relates to the global base or *limit* time, where $limit - base \leq 1$. Timestamp objects understand the join message which joins two timestamps as described in chapter 5. Figure 9-3 depicts the schema of event occurrence and timestamp definitions in the EVE database.

9.3.2 Event Detectors

Events are detected by persistent event detector objects which are part of the event detection and composition subsystem. After a new occurrence object is created, a message is sent to the appropriate event detector object. The detector object is reached through the event type object for the occurrence. After primitive event detection, composite event detection takes place. For that matter, an approach similar to that proposed for the centralized active database system Sentinel is used [Chakravarthy et al., 1994]. A composite event detector is a graph (see Figure 9-4), where nodes are event types and edges represent event composition. Nodes are marked with references to not-yet-consumed occurrences of component event types. Whenever an event is detected, the parent nodes are informed and check whether the new event together with already obtained component events can form a new event composition. This check is performed in an event type-dependent way. If multiple instances of a component event type exist, the oldest adequate one is chosen (this is the *chronicle* consumption mode [Chakravarthy et al., 1994]). In case the new event cannot be consumed, the detector stores a reference to it until a composition is possible, i.e., until new sibling component occurrences have been received. Event detection is recursive; whenever a parent node can compose a new occurrence, it in turn informs its parents, a.s.o. in a bottom-up manner. Event detector graphs may be distributed among different sites in which case, some event detector objects reference proxy detector objects which communicate with their “real” counterparts. In such cases, the EOB representing the remote server is informed of the occurrence and forwards the event over the network to the EVE server it represents.

In addition to event composition edges, event detector nodes reference rule objects. If a (primitive or composite) event occurrence is detected which has a rule attached to it, then the rule is added to the list of rules to be fired. This firing actually takes place as soon as the event detection cycle has been terminated. Rule execution is performed by the appropriate rule execution component.

9.3.3 Detector Synchronization

The accurate detection of time is an important issue in EVE. It is essential for the correct operation of a workflow system, that both *external* synchronization with the real world time and *internal* synchronization between individual workflow execution sites exists. In other words, the building of event timestamps must be meaningful with respect to real world events, i.e., physical clocks and the maintenance of temporal order are essential. The notion of time in distributed systems is problematic due to the fact that clock synchronization is achieved through the exchange of network messages whose travel duration is generally unpredictable. Physical clock synchronization is achieved by different protocols such as NTP (Network Time Protocol) which is the Internet clock synchronization standard [Mills 1991]. Although NTP on the Internet is implemented as a ‘best-effort’ service, the achievable synchronization error usually lies below 30 milliseconds even in wide area networks.

Event detection in EVE implements the composite event semantics defined in chapter 5. Each server host has a physical clock whose output can be read by appropriate software and scaled to suitable time units. ANSI C provides the `ctime()` function with which the system clock can be read. This function returns elapsed time in seconds since 00:00:00 UTC (Coordinated universal time), January 1, 1970. During event object creation, the event detection and composition subsystem of the EVE server uses this function to create a timestamp for this event, which effectively corresponds to a local clock tick.

For the correct composite event detection EVE server local clocks have to be synchronized with a precision P which is the maximum time difference between the corresponding ticks of any two server clocks. Thus, global time in EVE can be approximated by adjusting the granularity of each local clock to the global clock granularity g_g as defined in chapter 5. This global granularity must be larger than P in order to ensure that two simultaneous events receive timestamps distant at most one clock tick. The granularity requirements of the workflow application domain, usually in the order of seconds, lie comfortably above the precision achievable in the typical EVE operating environments.

Correct event composition in the chronicle consumption mode relies on the ordering of component occurrences using their timestamps. Upon event composition, the oldest eligible instance is chosen whenever there are multiple candidates. Thus, new events can only be composed if no older adequate component event has occurred at some site. This poses a special problem in EVE, as the signaling of events from remote sites typically takes different amounts of time or may even be temporarily impossible due to a detector site crash. Furthermore, the order of arrival of events from different remote sites may not be the same as their order of occurrence. Thus, an event detector can correctly compose new events out of components originating from multiple sites only if it is synchronized with the detectors of its components and so on recursively. Furthermore, efficiency considerations dictate that the effort to synchronize servers and their event detectors should not overly burden other workflow execution tasks. The efficiency of detector synchronization can be improved if we consider however, that only detectors on servers cooperating within some specific workflow instance need to be synchronized; Events generated within instances of unrelated workflow types are neither temporally nor causally related. Thus, synchronization is

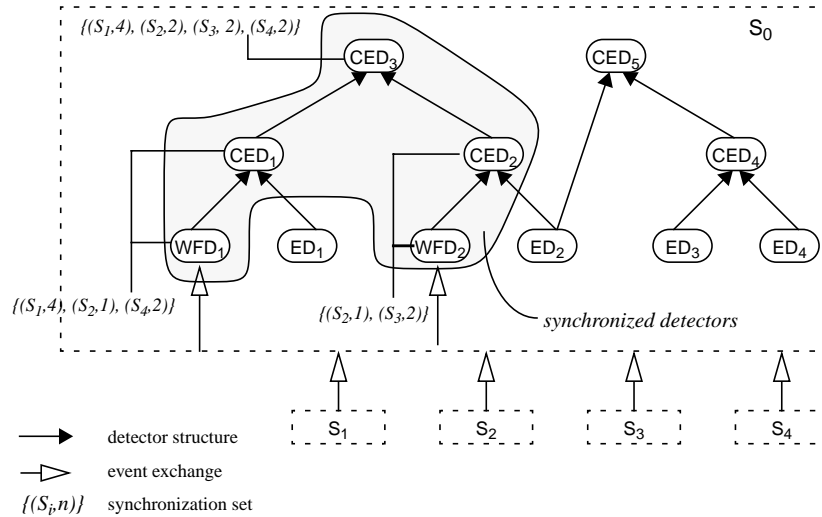


Figure 9-5: Synchronization of servers and event detectors in EVE.

needed for event detectors that detect workflow termination events or directly or indirectly contain such events as components.

The detector synchronization works as depicted in Figure 9-5. Whenever an EOB residing at a server S_0 requests a (sub)workflow and S_0 forwards this request to another server S_1 , then the affected detectors are informed that they need to synchronize with S_1 . The affected detectors are detectors of events whose component events may subsequently occur at different sites from that of the original request. Thus for example, assuming that a request WF1 has been forwarded from S_0 to S_1 , the affected detectors at S_0 include the detectors whose components include the event type reply WF1. Each of these detectors maintains the information on which servers it needs to synchronize (the *synchronization set*) with. In other words, the synchronization set of an event detector contains the servers of EOB which currently execute subworkflows of workflows executing at S_1 .

For each server S_i in the synchronization set of an event detector, a reference counter records the number of subworkflow instances requested by S_0 at S_i . It is also recorded for each server in the set when the last event originating at that server was received. Consequently the global time can be calculated until which the local server is synchronized with the involved remote servers (the *synchronization point*, which is the minimum of the events' timestamps least recently received from the remote servers). The server which has signalled this event is called the *most recently synchronized site*, MRS. Since it is guaranteed that all events with timestamps earlier than the synchronization point have been received, it is safe for a detector to consume these events for composite event detection. Whenever one of the affected detectors at S_0 receives an event or a component event, the timestamp of this event will be larger (i.e., the event has occurred later) than the synchronization point. In case the event was *not* sent by the MRS, the received event is queued and not further processed by the detector. Otherwise, the new MRS is determined, which then also determines the new synchronization point, and all queued events that occurred prior to the new synchronization point are flushed, i.e., processed as in centralized event detection.

Two kinds of events exchanged between servers are relevant for synchronization: *synchronization events* and *workflow termination events*. Synchronization events indicate that the originating server is still alive. They are sent out from each server at predefined synchronization intervals to all those servers from which request events for still executing (sub)workflow instances have been received. Thus, when a server S_0 receives a synchronization event from S_i , it is guaranteed

that it has received all previous workflow termination events from S_i , and can exploit this information for synchronization of its detectors.

Workflow termination events indicate the completion of a workflow instance; they are processed by event detectors. In case such an event is received, i.e., a workflow reply or exception, the reference counter of the respective server in the corresponding synchronization set is decreased. If the new value of the reference counter is 0, then no more instances of the corresponding workflow type are active at the remote server. This server can thus be deleted from the synchronization set of the respective detectors —and their ancestors. This procedure allows to minimize for each server and workflow type the number of remote servers it has to be synchronized with. Consequently, detectors at S_0 are only then synchronized with S_i if S_i in fact currently executes on behalf of S_0 one or more instances of the workflow type.

9.4 Run-Time Repository Organization

The REWORK run-time repository consists of localized data stored in EVE databases at each EVE server. The databases are implemented with the Shore object management system, described in the previous chapter. All EVE databases share a common schema divided into groups of related interfaces. These are the following:

- Types for the definition of operational workflow information. These include event detectors for each defined event type, workflow instances, timestamps, event occurrences, and error handling.
- Types to support of the workflow system administration environment. These include directory managers which support the efficient storage and retrieval of type objects and library managers which manage external compiled code.
- Types for the definition of EOB components. More specifically, operation types, rule types, run-time operation execution objects which maintain the state of operation execution by EVE clients, and state variable objects.
- Event notification rules are created for each capability defined in the REWORK system. These rules ensure that the appropriate EOB are informed of interesting event occurrences.
- A suitability defined for an event type is transformed into the condition of the ECA-rule defined for the event notification. For a created event occurrence, this condition computes the set of suitable EOB and checks for which of them the condition holds. It then applies the dispatch function specified in the responsibility and returns the resulting notification set of EOB for the occurrence. Ultimately, the action part of such a notification rule appends a copy of the reference to the event occurrence object to the EDI of the EOB determined in the condition part.

We consequently note that the abstractions provided by the REWORK metamodel have their direct counterparts implemented in the run-time repository used by EVE servers during workflow execution.

In general, the necessary run-time information is located in the database(s) of the server(s) where the information is needed. Complete EOB structure and task assignment information is required only at the database of the server to which the actor represented is connected. However, the event detectors of other servers must be aware of the actor location. A possible systematic optimization of information distribution in the run-time repository was not considered in the context

of this thesis. In the implemented prototype the entire run-time database was replicated among all server databases during system instantiating.

9.5 Event History Management

At a specific point in time, the sequence of event occurrences describes what has happened in the past. Each server maintains locally the events that were detected locally, including the events detected by synchronized detectors. These occurrences form the *event history* stored in Shore. First, the elements of the history together with other events occurring subsequently constrain what will happen in the future. This is the case for an occurrence if its type is a component of a composite event type and has not yet been consumed for an occurrence of that type. Second, (parts of) the entire event history may provide important information about EOB behavior and the course that individual workflow instances have taken. The event history thus is a database which can be used for the monitoring, analysis, and successive optimization of workflow types. The analysis of workflow execution histories lies beyond the scope of this thesis and is described in [Geppert & Tombros, 1997].

The history is maintained by EVE servers and is physically distributed among them. Each server maintains a consistent view of the global event history because composite events are inserted in it only after detector synchronization has taken place meaning that no earlier candidate component events have occurred. A logically integrated view of the global event history is possible based on the partial ordering of the timestamps of events.

9.6 Summary

In this chapter we described implementation aspects of a prototypical proof-of-concept platform for REWORK system execution. Various aspects relevant for production workflow execution, such as, security, recovery from server failure, and performance optimization, have not been considered in the context of this thesis. The implemented system however, has proven that event-based workflow execution is a feasible approach which provides a number of advantages.

The system is by its nature flexible and extensible. Every actor is represented by a separate Shore client application which is responsible for implementing the communication mechanisms necessary to interact with the actor. The configuration of the system, i.e., the description of the system components, is stored in the run-time repository thus allowing its modification during run-time by the creation of appropriate objects. The common infrastructure is kept as lightweight as possible and essentially consists of the event detection and composition subsystem, the message service subsystem, and the history management subsystem. Any necessary extensions to the basic functionality are implemented by appropriate actors.

The abstractions used in the REWORK metamodel have been directly implemented in the execution platform. Thus the operational semantics of the REWORK system are maintained by the resulting workflow system.

The fully distributed execution of workflows is supported by appropriate configuration of the system. The necessary trade-off that has to be made relates to the partitioning of workflow execution among servers, as increased partitioning necessitates increased synchronization traffic among the servers.

To conclude this chapter, we repeat the evaluation table presented in chapter 4, adding an evaluation of EVE's architectural quality attributes:

<i>System</i>	<i>Scalability</i>	<i>Availability</i>	<i>Modifiability</i>	<i>Integrability</i>
ProcessWEAVER	low	low	low	low
TriGS _{flow}	low	low	medium	low
COSA	low	low	low	medium
FlowMark	medium	low	low	medium
Meteor ₂	high	medium	low	medium
Mentor	medium	low	low	medium
MOBILE	medium	low	medium	high
WIDE	medium	medium	low	medium
Exotica/FMQM	high	low	low	medium
INCAs	high	low	low	?
EVE	medium	low/medium	high	high

10 The Workflow System Lifecycle Revisited

In this chapter we revisit the workflow system lifecycle introduced in chapter 1 and discuss how the set of concepts provided by the REWORK framework are put to use and how they affect the traditional ad hoc workflow system development process. We especially consider the phases of the proposed composition-based workflow system development process and discuss how each element of the framework affects the individual lifecycle tasks.

The present chapter is structured as follows: we initially discuss how the issues related to architectural mismatch [Garlan et al., 1995] are resolved in the REWORK framework. We subsequently present an overview of the proposed workflow system development and maintenance lifecycle. We subsequently describe the individual phases, the activities that take place in each phase and the relation between these activities and the elements of the REWORK metamodel.

10.1 Avoiding Architectural Mismatch with REWORK

The software architecture community has identified many of the difficulties encountered during the composition of large complex systems. An important set of problems has been subsumed under the term of '*architectural mismatch*' [Garlan et al., 1995]. Architectural mismatch occurs because the components of the composed system may make conflicting assumptions about different aspects of the composed system:

- assumptions concerning the *nature of components* include those about the infrastructure on which the components are built, about the used control model, and about the underlying data model of the composed system;
- assumptions concerning the *nature of connectors* refer to the communication and coordination protocols between components and about the data that is being communicated over connectors;
- assumptions concerning the *global architectural structure* refer to cooperation assumptions between participating components; and
- assumptions concerning the *construction process* of the system include the kind of code used and the build-process dependencies.

These issues are by definition relevant in a system construction framework such as REWORK where the basic assumption is that a workflow system is composed out of heterogeneous actors. Consequently it has been an important goal of our work to solve the architectural mismatch problem. The solution essentially consists of the following aspects:

- The REWORK framework provides a unique infrastructure for the operation of actors — EVE. Furthermore, due to the homogeneous nature of EOB which represent these actors, a single control model is provided for every component which abstracts from eventual differences of the control model of participating actors.

- Assumptions about other components in the system are made explicit through the use of the service abstractions and task assignment policies. Whatever functionality is available in a REWORK system is coded by an appropriate service definition which can be examined in the system repository.
- A global architectural structure is provided which determines what kind of components exist in the composed system, how they are integrated, and what kinds of interactions are possible.
- Design guidance is provided by the REWORK framework for both the analysis of new actor types and the subsequent composition of the workflow system based on the predefined properties of EOB used for their integration.

It is however imperative, that the effectiveness of the REWORK framework in eliminating the architectural mismatch between heterogeneous application systems is tested under a production environment. The resources available in the context of this thesis limit the possibility for an extensive and long-term evaluation. Furthermore, it is expected that for certain types of information systems, a complete and seamless integration in a REWORK system might not be possible. This especially concerns monolithic systems which make a closed world assumption and do not provide adequate external access points.

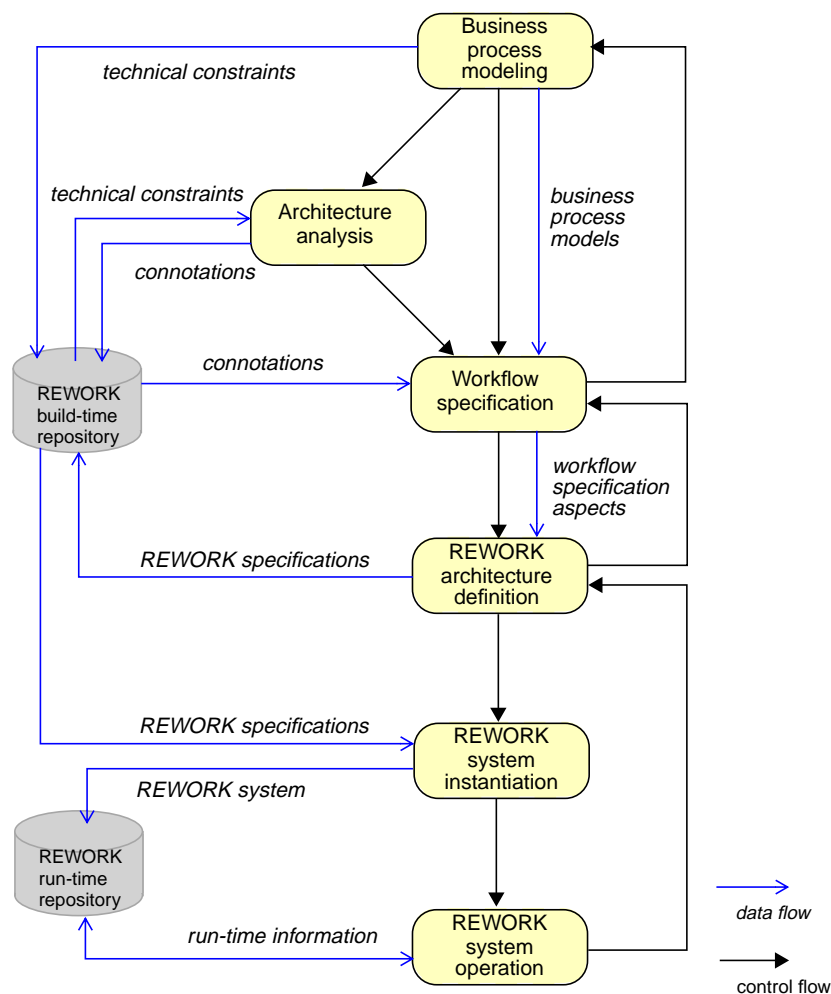


Figure 10-1: The REWORK system development lifecycle. Boxes denote lifecycle phases, arrows denote informational and causal dependencies between phases.

10.2 The REWORK System Development Lifecycle

Figure 1-1 depicts the phases of the REWORK system development lifecycle. We are now in a position to overview the entire lifecycle supported by the REWORK framework and describe how each of the concepts developed in this thesis fits into the greater picture. We note however, that in our work we do not cover the phases related to business process modeling. Furthermore, we consider the workflow specification phase only with respect to its effect on the REWORK architecture definition phase and to its integration in the REWORK development environment.

10.2.1 Preliminaries and Business Process Modeling

The basic assumption underlying the proposed lifecycle is that a project initiation phase precedes the entire lifecycle. In this phase, the goals with respect to the developed class of workflow systems are set and the project team is formed (including the roles of the workflow system architect and workflow system composer—see chapter 4). The project management team determines which business processes will be automated by the developed workflow systems. An appropriate business process modeling (BPM) formalism should be selected at this point. In our work we have strived to provide a neutral and flexible architecture and implementation substrate with respect to the BPM formalism. It is important however, that a bridge is provided by the BPM environment to a workflow specification language which can be mapped to the concepts of the REWORK metamodel. Alternatively, it is conceivable that high-level business process specifications are directly implemented by the REWORK metamodel. In that case, the semantic gap is larger and should be taken into consideration. This issue is however, beyond the scope of our work.

Once an appropriate BPM formalism is chosen, the selected business processes are modeled as-is, including organizational and technical information. Technical information should be taken into account in order to identify as early as possible, existing restrictions of the applications and information systems to be integrated. During this phase, the relevant organizational structure and eventually the types and sources of documents which will be forwarded in the system should be analyzed.

10.2.2 Architecture Analysis

During architecture analysis, the technical information and constraints pertaining to the applications and information systems which are integrated in the developed REWORK systems are considered. The main conceptual instrument used in this phase is the analysis and classification framework described in chapter 4. This phase is initiated in the following cases:

- the identification of involved information systems in the business processes which are to be automated has been completed;
- the business process modeling phase has been completed;
- the organization is making an inventory of existing information systems and applications. In this case, the actors identified and characterized may be integrated in the future.

The characterization of the information systems which have been identified is based on actor connotations. The workflow system architect examines the available technical documentation, interviews developers and current users, and considers information regarding present integration mechanisms, with the goal of determining the terms of participation and implementation attributes of the actor. Once these values are determined, they are stored in the build-time repository and can be used in the subsequent design phase to provide access paths to appropriate design

artifacts representing the actor in REWORK specifications. Furthermore, during the analysis phase, the kinds of organizational relationships and organizational units must be identified which are particular to the currently developed workflow system. The need to define new organizational relationship types may subsequently arise. The advantages of this approach are twofold:

- there is a clear separation of the analysis and design phase of the workflow system and information system integration architecture; and
- the reuse potential of the classification increases as more systems are developed. The acquired expertise facilitates the subsequent analysis in other projects.

10.2.3 Workflow Specification

The workflow specification phase requires as its input the as-is business process model and the connotations of the participating actors. It produces a description of the to-be process models in the form of a *source workflow specification* which will be integrated in the developed REWORK system (see chapter 7). The automatic transformation of a business process model to a workflow specification is usually not feasible [Weske et al., 1999]. A complete source workflow specification must describe the to-be process structure, task dependencies, data flows between tasks, the organizational model, and technical infrastructure and resources needed by workflow tasks. It is of course possible that the resulting workflow specification may represent multiple process models as e.g., in TRAMs [Kradolfer & Geppert, 1997].

The main constraint from the perspective of REWORK is that the source workflow specification must be expressed in a modeling language whose metamodel can be mapped to the REWORK metamodel. As already demonstrated in chapter 7, a large class of activity-based modeling formalisms can be mapped to the abstractions provided by the REWORK metamodel in a straightforward way. Depending however on the expressiveness of the chosen source workflow metamodel, as well as on existing expertise and preferences in the development organization, multiple formalisms may be used simultaneously to express the various aspects of the developed workflow system. Furthermore, it is conceivable that a high-level workflow specification is only used for modeling certain aspects of the developed workflow system. For example, solely the data flow aspects are specified by a developer who is already familiar with a particular data modeling language. The REWORK metamodel abstractions must then be used for describing the remaining aspects during the subsequent architecture definition phase.

When multiple languages are used for process modeling, data modeling and organizational modeling, the source metamodels have to be mapped to the REWORK metamodel in such a way as to complement each other. In certain cases, and depending on the orthogonality of the representation of these aspects in the source workflow metamodel, this may mean that only part of the source metamodel abstractions may be used. The problems inherent in the use of multiple source workflow specification languages have to be solved on a case-by-case basis and are a potential subject of further work. A solution to this problem probably involves the definition of a classification framework for workflow metamodels which will allow the reasoning about the consistency and orthogonality of modeling abstractions.

After workflow specification, a review activity may be performed in order to check its completeness and/or correctness —subject of course to the specification method supporting such checks. This may determine that further information may be required from the architecture analysis and/or business process modeling phase. This will lead to an iteration over the business process modeling/architecture analysis activities.

10.2.4 REWORK Architecture Definition

The phase following the workflow specification in a source workflow language involves the development of a homogeneous architectural model of the implemented workflow system. The activity is performed by the workflow system composer and is supported by browsing, editing and consistency checking tools of the REWORK build-time repository. The input of this activity consists of the workflow specification aspects prepared during the previous phase. The various elements of the architecture definition are summarized in Table 10-1, where the effective correspondence between relevant facts from the real world, their respective aspect in workflow specification, and finally the provided REWORK abstraction is depicted. A resulting complete REWORK system architecture includes the appropriate abstractions which of course have to pertain to the intended static structure and dynamic behavior of the developed workflow system. Furthermore, the invariants defined for REWORK systems are expressed by appropriate integrity constraints over the repository objects.

Table 10-1: REWORK architecture definition elements

<i>Real-world fact</i>	<i>Workflow specification aspect</i>	<i>REWORK abstraction</i>
information system components and connectors	workflow system architectural structure	EOB, EVE, event type registration (sections 6.1, 6.2)
organizational constraints under which the workflow is executed	organizational structure	organizational relations (section 6.3)
actor functionality used in workflow	workflow tasks	services (section 5.2)
activity execution sequence	control flow	ECA-rules (section 7.2)
activity data dependencies	data flow	event parameters (section 5.3)
business rules	task assignment	suitability, dispatching (section 6.4)
human role, application functionality	actor behavior	ECA-rules, rule packages (section 6.2)

10.2.5 REWORK System Instantiation and Operation

The concluding phase, prior to the operation of a developed REWORK system, comprises the instantiation of the REWORK specification. As described in chapter 9, this activity involves the creation of objects in the REWORK run-time repository distributed over the EVE server databases. These objects include EOB aggregates, EVE site objects, event detectors, rule-management and execution objects, as well as administrative objects.

The distributed architecture of REWORK systems must be taken into consideration during the instantiation process. Prior to REWORK system operation, the entire collection of run-time objects must be generated at the appropriate participating sites. The instantiation process is thus guided by a global protocol which executes at a REWORK development site and starts local transactions in the participating EVE server databases (see Figure 9-1). In the event of failure of a local transaction, this transaction has to be executed again at some later time, the remaining local transactions however, are not automatically rolled back. The approach, consequently, requires the definition of appropriate compensation transactions in case the instantiation process has to be aborted in its entirety. We also mention at this point that the loose coupling of REWORK components allows —with certain restrictions— the addition of new components after the initial system

instantiation. These restrictions in particular, and REWORK system evolution in general, are the subject of further work.

Following a successful system instantiation multiple instances of the defined processes can start executing following the generation of the initial request event. Each process execution results in an event history, as described in chapter 6. We finally mention at this point that changing environmental factors as well as observations pertaining to the operation of a REWORK system (e.g., performance bottlenecks) may lead to an architectural redesign.

10.3 Summary

In this chapter we considered how the use of the REWORK development framework can help reduce the architectural mismatch problem which typically occurs when integrating heterogeneous information systems. We also described the activities composing the lifecycle of REWORK systems and the required input and output of each activity.

11 Conclusion

In this chapter we conclude this thesis by summarizing the main issues considered and the contributions made. We also identify some directions for future work.

11.1 Contributions

In this thesis we described the REWORK event- and repository-based framework for workflow system architecture. The framework provides an appropriate metamodel (REWORK) for the description of structure and behavior of reactive workflow system components and ultimately, entire workflow systems. The components of an operative system are stored in a run-time repository and interact during workflow execution through an event-based mechanism. In order to realize this framework, we have developed a novel approach to leverage active database technology to distributed workflow enactment.

The explicit view of workflow system architecture provided by the REWORK metamodel has the important advantage of allowing the extension of the WFMS functionality by constructing and integrating appropriate components. This compares favorably to most other approaches which do not provide mechanisms to explicitly describe the workflow execution infrastructure.

An important aspect of the REWORK metamodel concerns the formalization of the operational semantics of the developed workflow systems. The observable behavior of the REWORK components is expressed by the events that are stored in the event history created during workflow execution. We defined the notion of a correct component behavior over an event history, based upon which, the notion of workflow execution correctness can be expressed.

Furthermore, we considered the problem of repository-based workflow system composition. The complexity of the construction of workflow systems can be reduced if a methodical composition approach is used, supported by the appropriate conceptual abstractions and software infrastructure. In this thesis we do not provide a workflow system development method spanning the entire workflow system lifecycle. Rather, we concentrate on the description of an intermediate architectural metamodel for workflow systems onto which more abstract specification languages can be mapped. In particular, the reuse of architectural artifacts —components and connectors— is advocated in our work. We described the requirements, the structure, and a possible implementation of a build-time repository for the architectural workflow system artifacts.

The analysis of workflow management system infrastructure and the actors participating in workflow execution, as described in chapter 4, provides the conceptual basis for the development of the REWORK metamodel for the description of workflow system components. This analysis identified basic attributes of workflow actors which are important for their characterization and integration, and which are ultimately mapped to appropriate workflow system component implementations.

The REWORK metamodel proposed in chapters 5 and 6 provides the appropriate abstractions for the repository-based composition of workflow systems and the definition of operational se-

manatics for workflow execution. The event-based approach towards workflow system integration which is supported by the metamodel, provides important advantages for the flexible integration of components. Events provide a uniform mechanism for both the synchronous and asynchronous interaction of components simplifying the necessary interaction and coordination infrastructure. The result is a relatively light-weight and configurable system footprint.

The use of events for the expression of workflow situations and ECA-rules for inter-task dependencies is a powerful —albeit low-level— paradigm which can be used to represent different kinds of workflow specification approaches. In chapter 7, we have shown how the broad class of activity-based workflow specification languages can be mapped to the constructs provided by the REWORK metamodel. The mapping allows the operationalization of different specification approaches which can be tailored to the requirements of the concrete workflow application system being developed, but can make use nevertheless of a common underlying operational architecture.

In chapter 8, we have described a repository for the storage of workflow system architectural artifacts and discussed how these artifacts may be retrieved and reused for workflow system composition. In chapter 9, we presented a possible implementation for an operational platform for REWORK system. The described system directly implements the concepts described in the preceding chapters and is intended as a proof-of-concept for the proposed solution. In chapter 10 we concluded by providing an overview of the REWORK lifecycle for workflow system composition and operation.

Summarizing, this thesis contributes to the description of workflow systems at an architectural level. It also elaborates an event/ECA-rule-based architectural style for workflow systems. Finally, it provides abstractions and repository-based mechanisms for the reuse of architectural artifacts for workflow system composition.

11.2 Directions for Future Work

Due to the rather broad spectrum of issues, several issues could not be addressed in this thesis and some restrictive assumptions and simplifications had to be made.

The focus of our work has been on the definition of an operational architecture for workflow systems. We have thus not investigated in depth the phases preceding and succeeding the definition of workflow system architecture. The specification of the workflow applications, the optimization of the resulting system, and most importantly the maintenance and evolution of workflow system architecture have been considered only superficially. Our work has concentrated on providing a framework for systematic workflow system composition but has not focused on run-time efficiency, security, and scalability issues. These issues are of increasing importance for large-scale workflow system implementation.

While the REWORK metamodel is powerful enough to describe workflow systems which consider atomic workflow activities as black boxes, there has been some recent research in related fields which may indicate that this is not always feasible, especially when heterogeneous process support systems have to cooperate. In such cases, interaction between these systems may take place during the execution of individual activities. Although in this thesis the semantics of workflow execution have been defined under the assumption that workflow system components are black-boxes with respect to the execution of individual services, it is conceivable that the operational semantics can be extended to consider the execution, for example, of rule actions and further interaction event types, thus paving the way to considering intermediate execution states of

individual activities. Thus, extensions of this work for building heterogeneous workflow system federations may be necessary.

Another subject for future work is the development of a user-friendly workflow system composition environment. The implemented repository and the supporting tools have been developed in order to prove the basic feasibility of the approach. The effort required for developing a fully-featured environment would clearly surpass the time and manpower limits available within the context of a single thesis. Once however such an environment is available, experience with its operation and performance can be valuable in successive optimization. Furthermore, only repeated use and extensive population of the architectural repository will begin to payoff the initial development effort.

A final issue concerns the improvement of the underlying coordination system. This relates more to the underlying storage management system and less to the messaging system which is based on the relatively stable platform of ACE. However, the object management system used for the REWORK execution platform is only an prototype clearly lacking the robustness and performance of industrial-strength object management systems. Thus it is our opinion, that an appropriate commercial OODBMS should be used in the future.

Still, many issues remain open regarding the engineering-wise development of workflow systems. It is our hope, that the results of this thesis will contribute to a compositional approach towards workflow system construction.

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
ACTMAN	Actor Manager
ADBS	Active Database System
API	Application Programming Interface
ATM	Advanced Transaction Model
BPR	Business Process Reengineering
CASE	Computer Aided Software Engineering
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DCOM	Microsoft Distributed Component Object Model
DDL	Data Definition Language
DML	Data Manipulation Language
ECA	Event Condition Action
ECAMAN	ECA-Rule Manager
EDI	Event Delivery Interface
EOB	Event Occurrence Broker
EPI	Event Posting Interface
EVE	Event Engine
GUI	Graphical User Interface
IDL	Interface Definition Language
IT	Information Technology
JDBC	Java Database Connectivity
LRU	Least Recently Used
ODBC	Open Database Connectivity
ODMG	Object Data Management Group
ODL	Object Definition Language
OLE	Object Linking and Embedding
OLTP	Online Transaction Processing
OMA	Object Management Architecture
OMG	Object Management Group
OODB	Object-oriented Database System
ORB	Object Request Broker
OQL	Object Query Language
OS	Operating System
PCTE	Portable Common Tool Environment
REWORK	Reactive Workflow Component
SDL	Shore Data Language

SEE	Software Engineering Environment
SQL	Structured Query Language
STATEMAN	State Manager
WAPI	Workflow Application Programming and Process Interchange Interface
WARIA	Workflow and Reengineering International Association
WAD	Workflow Activity Definition
WPD	Workflow Process Definition
WfMC	Workflow Management Coalition
WFMS	Workflow Management System
WM	Workflow Management
WSL	Workflow Specification Language

Glossary

The following glossary provides a brief summary of the terminology used in this thesis. The words preceded by an \rightarrow reference further entries in the glossary.

Actor

An entity requesting and providing services in a \rightarrow workflow system. It can be either a software system, a human being, or an organizational entity. Synonymous to processing entity.

Architecture

The overall design of a software system. An architecture integrates various aspects of a system and defines guidelines which can be used during system composition.

Black box integration

A tool or application which provides no API or other access interface is integrated by means of wrappers which perform transformation of data and control information between the integration environment and the black box. See also \rightarrow white box integration.

Component

A component is a unit of composition with well-defined interfaces and explicit context dependencies. In the \rightarrow REWORK metamodel context dependencies are specified by describing the behavior of a component.

Component instance

A simplifying notion. \rightarrow Components as such do not have direct instances. However, components in general and \rightarrow event occurrence brokers in particular, are templates composed of instantiable abstractions such as \rightarrow ECA-rules, state variables, etc. A component instance is an aggregate object composed of instances of these abstractions.

Composite event type

An \rightarrow event type composed out of further event types by the application of one or more of the following operators: conjunction, exclusive-or disjunction, inclusive-or disjunction, sequence, concurrency, iteration, negation.

Composition

Assembly of components into a whole without modifying the participating components. The semantics of the composite can be derived from those of the components.

Event

An asynchronous occurrence raised by a component in an instantiated \rightarrow REWORK

system. Events are propagated using messages or event objects. An event propagator travels from the event source \rightarrow component instance to the interested event sink component instances. Event occurrences have a corresponding \rightarrow event type.

Event Condition Action Rule (ECA-rule)

An Event Condition Action rule (or trigger) is an abstraction for the specification of reactive behavior. In its most general form, an ECA-rule consists of the following parts: an \rightarrow event type meaning that whenever events of this type occur the rule will be triggered; a condition specification which is checked when the rule is triggered; an action specification which defines the actions that are performed when the rule is triggered and the condition evaluates to true.

Event engine (EVE)

A distributed event notification and composition platform which underlies every instantiated \rightarrow REWORK system. An event engine is required to integrate and coordinate *event occurrence broker* instances.

Event history

The trace of \rightarrow events which is recorded when \rightarrow workflows are executed by an instantiated \rightarrow REWORK system. The formal semantics of workflow execution can be formalized in terms of permissible event histories.

Event occurrence

\rightarrow event.

Event occurrence broker (EOB)

A \rightarrow component used in an a \rightarrow REWORK system to represent/encapsulate one or more \rightarrow processing entities. An EOB has predefined properties, makes available a set of \rightarrow services, and supports specific interactions with the represented processing entities. An EOB is composed out of predefined subcomponents which communicate through a shared message bus. EOB \rightarrow component instances are coordinated by means of a distributed \rightarrow event engine.

Event type

A template for the instantiation of \rightarrow events. An event type can be \rightarrow primitive or \rightarrow composite. The type determines the semantics of its occurrences, i.e., how occurrences are detected, what are the permissible event parameters, and what are the possible event type compositions in which the type can participate.

Framework

A collection of reusable design decisions about a specific software domain. It can consist of classes and other abstractions as well as of descriptions of their collaboration patterns.

Glass box

A component or module whose implementation and consequently its internal operation mode is visible, but which cannot be modified by its users.

Interface

The conceptual limit of a software entity which supports a predefined interaction protocol.

Object

An entity that comprises state and behavior and has a unique identity.

Primitive event type

Any of the following \rightarrow event types: time, service request, service reply, service exception, request confirmation.

Processing entity

\rightarrow actor

REWORK metamodel

A metamodel providing the abstractions and linguistic constructs for the specification of the software architecture of \rightarrow workflow systems. The REWORK metamodel is the basis for the specification of an instantiable \rightarrow REWORK system.

REWORK specification

An REWORK metamodel-compliant specification defining the software architecture of a \rightarrow workflow system. It comprises the specification of \rightarrow event occurrence brokers representing the \rightarrow processing entities which operate in the workflow system, the types of the permissible \rightarrow event occurrences, the organizational relationships between event occurrence brokers, and functions determining task assignment strategies.

REWORK system

An instantiated \rightarrow REWORK specification executable on top of an \rightarrow event engine.

Service

A unit of functionality that is provided by a \rightarrow processing entity in an \rightarrow REWORK system.

WfMC Reference Architecture

A reference model developed by the Workflow Management Coalition which describes the main functional subsystems of \rightarrow workflow systems and the \rightarrow interfaces between these subsystems. For a more detailed description and an evaluation see section 3.4.1 of this thesis.

White box integration

A component or module is integrated by means of modification to its implementation which is accessible to system integrators. See \rightarrow black box integration and \rightarrow glass box.

Workflow

A \rightarrow workflow is a collection of tasks which are performed by software systems, people and groups of people, or a combination of both. A workflow is executed according to its specification.

Workflow management system (WFMS)

A workflow management system is a system that completely defines, manages, and executes \rightarrow workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic.

Workflow metamodel

A conceptual framework providing the concepts and semantics for the definition of workflow models. One or more \rightarrow workflow specification languages can be defined to express the concepts of the metamodel by specific linguistic mechanisms.

Workflow model

A description of a specific \rightarrow workflow process in a \rightarrow workflow specification language. Synonymous to workflow process definition.

Workflow process definition

\rightarrow workflow model.

Workflow specification

\rightarrow workflow model.

Workflow specification language

A system of linguistic abstractions and mechanisms compliant to a \rightarrow workflow metamodel.

Workflow system

A workflow system is an information system targeted towards the execution of a set of defined workflows. It consists of a workflow management system and the integrated \rightarrow processing entities composing the workflow application system (or simply workflow application) executing workflow tasks.

References

Abbot & Sarin, 1994

K. Abbot, S. Sarin. Experiences with workflow management: issues for the next generation. In *Proceedings 1994 ACM Conference on Computer Supported Cooperative Work*, (Chapel-Hill, NC). ACM Press.

Abiteboul et al., 1995

S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases*. Reading, MA (etc.): Addison-Wesley, 1995.

Abowd et al., 1995

G.D. Abowd, R. Allen, D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4), pp. 319–364, October 1995.

Allen & Garlan, 1997

R. Allen, D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), pp. 213–249, July 1997.

Alonso et al., 1995

G. Alonso, C. Mohan, R. Günthör, D. Agrawal, A. El Abbadi, M. Kamath. Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management. In *Proceedings IFIP Working Conference on Information Systems for Decentralized Organizations*, (Trondheim, Norway, August 1995).

Alonso & Schek, 1996

G. Alonso, H.-J. Schek. Research Issues in Large Workflow Management Systems. In *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems* (Athens, GA, May 1996).

Alonso et al., 1997

G. Alonso, C. Hagen, H.-J. Schek, M. Tresch. Distributed Processing over Stand-Alone Systems and Applications. In *Proceedings 23rd VLDB Conference*, (Athens, Greece, August 1997). Morgan Kaufmann.

Alonso et al., 1997a

G. Alonso, D. Agrawal, A. El Abbadi, C. Mohan. Functionality and Limitations of Current Workflow Management Systems. http://www.almaden.ibm.com/cs/exotica/exotica_papers.html, IBM Technical Report, Almaden Research Lab, 1997.

Apple Computer, 1991

Apple Computer Inc. *Inside Macintosh, Volume VI*. Addison-Wesley, 1991.

Arango, 1991

G. Arango. Domain Analysis – From Art Form to Engineering Discipline. In R. Prieto-Díaz, G. Arango (eds), *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, 1991.

Armenise et al., 1993

P. Armenise, S. Bandinelli, C. Ghezzi, A. Morzenti. A Survey and Assessment of Software

Process Representation Formalisms. *International Journal on Software Engineering and Knowledge Engineering*, 3(3), pp. 401–426, 1993.

Atkinson et al., 1992

M. Atkinson, F. Bancilhon, D. DeWitt, K.R. Dittrich, D. Maier, S. Zdonik. The Object-Oriented Database System Manifesto. In F. Bancilhon, C. Delobel, P. Kanellakis (eds.), *Building an Object-Oriented Database System: The Story of O₂*. Morgan Kauffman, 1992.

Atkinson et al., 1996

M. Atkinson, L. Daynès, M. Jordan, T. Printezis, S. Spence. An Orthogonally Persistent Java. *ACM SIGMOD Record*, 25(4), pp. 68–75, December 1996.

Babaoglou & Marzullo, 1993

O. Babaoglou, K. Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In S. Mullender (ed.), *Distributed Systems*. Addison-Wesley/ACM Press, 1993.

Barbara et al., 1996

D. Barbara, S. Mehrotra, M. Rusinkiewicz. INCAs: Managing Dynamic Workflows in Distributed Environments. *Journal of Database Management: Special Issue on Multidatabases*, 7(1), Winter 1996.

Barghouti et al., 1996

N. S. Barghouti, W. Emmerich, W. Schäfer, A. H. Skarra. Information Management in Process-Centered Software Engineering Environments. In A. Fuggetta, A. Wolf (eds.), *Software Process*. John Wiley & Sons, 1996.

Barret et al., 1996

D. J. Barrett, L. A. Clark, P. L. Tarr, A. E. Wise. A Framework for Event-Based Software Integration. *IEEE Transactions on Software Engineering Methodology*, 5(4), pp. 378–421, October 1996.

Bandinelli et al., 1996

S. Bandinelli, E. Di Nitto, A. Fuggetta. Supporting cooperation in the SPADE-1 environment. *IEEE Transactions on Software Engineering*, 22(12), pp. 841–865, December 1996.

Bass et al., 1998

L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

Belkhatir et al., 1994

N. Belkhatir, J. Estublier, W. Melo. ADELE-TEMPO: An Environment to Support Process Modeling and Enaction. In J.-C. Derniame, A. Finkelstein, J. Kramer, and B. Nuseibeh (eds.), *Software Process Modeling and Technology*. John Wiley & Sons, 1994.

Ben-Shaul & Kaiser, 1995

I. Ben-Shaul, G. Kaiser, G. Heineman. *A Paradigm for Decentralized process Modeling*. Kluwer Academic Publishers, 1995.

Berndtsson et al., 1997

M. Berndtsson, S. Chakravarthy, B. Lings. Task sharing among agents using reactive rules. *Proceedings 2nd IFCIS International Conference on Cooperative Information Systems (Charleston, South Carolina, USA, June 1997)*. IEEE Computer Society Press.

Bernstein et al., 1987

P. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

Bernstein, 1996

P. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2), pp. 86–98, February 1996.

Bernstein, 1998

P. Bernstein. Repositories and Object-Oriented Databases. *SIGMOD Record*, 27(1), pp. 88–96, March 1998.

Bleicher, 1991

K. Bleicher. *Organisation: Strategien – Strukturen – Kulturen*. Gabler, 1991.

Boehm, 1988

B. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(2), pp. 61–72, May 1988.

Boehm & Sherlis, 1992

B. Boehm, W. Sherlis. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference (Arlington, VA, April 1992)*.

Bogia & Kaplan, 1995

D.P. Bogia, S.M. Kaplan. Flexibility and Control for Dynamic Workflows in the wOrlds Environment. *Proceedings Conference on Organizational Computing Systems, (Milpitas, CA, November 1995)*. ACM Press.

Böhm et al., 1998

A. Böhm, H. Rieseler, W. Sengler. Bürokommunikation und Workflow auf der Basis von Microsoft Exchange. In W. Köhler-Frost (eds.), *Electronic Office Systeme*. Erich Schmidt Verlag, 1998.

Booch et al., 1999

G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

Borghoff et al., 1997

U. Borghoff, P. Bottoni, P. Mussio, R. Pareschi. Reflective Agents for Adaptive Workflows. *Proceedings 2nd Int'l Conf. on the Practical Application of Intelligent Agents and Multiagent Technology*, London, UK, April 1997.

Buchmann et al., 1995

A. Buchmann, J. Zimmermann, J. Blakeley, D. Wells. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proceedings of the Eleventh International Conference on Data Engineering, (March 6-10, 1995, Taipei, Taiwan)*. IEEE Computer Society Press.

Buschmann et al., 1996

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

Bussler & Jablonski, 1994

C. Bussler, S. Jablonski. Implementing Agent Coordination for Workflow Management Sys-

tems Using Active Database Systems. In *Proceedings Research Issues in Data Engineering*, (Houston, TX, February 1994). IEEE Computer Society Press.

Bussler, 1997

C. Bussler. *Organisationsverwaltung in Workflow Management Systemen*. Doctoral Dissertation, University of Erlangen, March 1997.

Cagan, 1990

M. Cagan. *The HP SoftBench Environment: An Architecture for a New Generation of Software Tools*. Hewlett-Packard Journal, 41(3), pp. 36-47, June 1990.

Carey et al., 1994

M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, M. J. Zwillig. Shoring Up Persistent Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (Minneapolis, Minnesota, May 24-27, 1994)*. *SIGMOD Record*, 23(2), June 1994.

Casati et al., 1995

F. Casati, S. Ceri, B. Pernici, G. Pozzi. Conceptual Modeling of Workflows. In *Proceedings Object-Oriented and Entity Relationship Approach '95 International Conference*, (Gold Coast, Australia, Dec. 12-15, 1995). Springer Verlag.

Casati et al., 1996

F. Casati, S. Ceri, B. Pernici, G. Pozzi. Deriving Active Rules for Workflow Enactment, *Proceedings 7th International Conference on Database and Expert System Applications*, (Zurich, Switzerland, September 1996). Springer Verlag.

Cattell et al., 1997

R. G. G. Cattell, D. K. Barry et al. (eds.) *The Object Database Standard: ODMG 2.0*. Morgan Kauffman, 1997.

Ceri et al., 1997

S. Ceri, P. Grefen, G. Sanchez. WIDE – A Distributed Architecture for workflow Management. *Proceedings 7th International Workshop on Research Issues in Data Engineering (Birmingham, England, April 1997)*.

Chakravarthy et al., 1994

S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts, and Detection. *Proceedings 20th International Conference on Very Large Data Bases*, (Santiago, Chile, September 1994).

Chakravarthy & Misra, 1994

S. Chakravarthy, D. Misra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(1), pp. 1–26, November 1994.

Chen & Dayal, 1996

Q. Chen, U. Dayal. A Transactional Nested Process Management System. *Proceedings 12th International Conference on Data Engineering (New Orleans, Louisiana, February 26 - March 1, 1996)*. IEEE Computer Society Press.

Constantopoulos & Dörr, 1995

P. Constantopoulos, M. Dörr. Component Classification in the Software Information Base. *Object-Oriented Software Composition*. Prentice Hall, 1995.

Cook & Wolf, 1995

J.E. Cook, A. Wolf. Automating Process Discovery through Event-Data Analysis. *Proc. 17th International Conference on Software Engineering*, (April 1995). ACM Press.

COSA, 1998

COSA Workflow 2.0 Product Specification. <http://www.cose.de/>, February 1998.

CSE, 1996

CSE Systems, Distributed^{WF} CSE/Workflow. <http://www.csesys.co.at/>, August 1996.

Curtis et al., 1992

B. Curtis, M.I. Kellner, J. Over. Process Modelling. *Communications of the ACM*, 35(9), pp. 75–90, September 1992.

Dami et al., 1998

S. Dami, J. Estublier, M. Amiour. APEL: a Graphical Yet Executable Formalism for Process Modeling. In E. Di Nitto, A. Fuggeta (eds.), *Process Technology*. Kluwer Academic Publishers, 1998.

Dayal et al., 1990

U. Dayal, M. Hsu, R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (Atlantic City, NJ, May 23-25, 1990), *SIGMOD Record*, 19(2), June 1990.

Davenport & Short, 1990

T.H. Davenport, J.E. Short. The New Industrial Engineering: Information Technology and Business Process Redesign. *Sloan Management Review*, Summer 1990.

Davenport, 1993

T.H. Davenport. *Process Innovation: Reengineering Work through Information Technology*. Harvard Business School Press, 1993.

Dennings, 1997

A. Denning. *ActiveX Controls Inside Out*. Microsoft Press, 1997.

Derungs, 1996

M. Derungs. Vom Geschäftsprozess zum Workflow. In [Österle & Vogler, 1996].

Dittrich et al., 1987

K. R. Dittrich, W. Gotthard, P. C. Lockemann. DAMOKLES - The Database System for the UNIBASE Software Engineering Environment. *Data Engineering Bulletin*, 10(1), pp. 37–47, January 1987.

Dittrich et al., 1995

K.R. Dittrich, S. Gatzui, A. Geppert (eds). The Active Database Management System Manifesto. A Rulebase of ADBMS Features. In T. Sellis (ed.) *Proceedings 2nd International Workshop on Rules in Databases (Athens, Greece, September 1995)*. Springer Verlag.

Dittrich & Gatzui, 1996

K. R. Dittrich, S. Gatzui. *Aktive Datenbanksysteme: Konzepte und Mechanismen*. International Thomson Publishing, 1996.

Dowson, 1987

M. Dowson. Integrated Project Support with IStar. *IEEE Software*, 4:6, November 1987.

Drucker, 1988

P. Drucker. The Coming of the New Organizations. *Harvard Business Review*, 1, January-February 1988.

Eder & Groiss, 1996

J. Eder, H. Groiss. Ein Workflow-Management-System auf der Basis aktiver Datenbanken. In G. Vossen, J. Becker, *Geschäftsprozeßmodellierung und Workflow-Management: Modelle, Methoden, Werkzeuge*. International Thomson Publishing, 1996.

Elmagarmid, 1992

A. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.

Elmasri & Navathe, 1989

R. Elmasri, S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

Fernstrøm, 1993

C. Fernstrøm. Process Weaver: Adding Process Support to UNIX. In *Proceedings of the 2nd International Conference on the Software Process – Continuous Software Process Improvement (1993)*. IEEE Computer Society Press.

FileNet, 1998

FileNet Visual WorkFlo. <http://www.filenet.com/>, April 1998.

Fischer, 1996

G. Fischer. Domain Oriented Design Environments. In *Proceedings 18th International Conference on Software Engineering, (Berlin, Germany, March 25-29, 1996)*. IEEE Computer Society Press.

Gaitanides, 1994

M. Gaitanides (ed.). *Prozessmanagement: Konzepte, Umsetzungen, und Erfahrungen*. Hanser Verlag, 1994.

Gamma et al., 1995

E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Garlan & Notkin, 1991

D. Garlan, D. Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. In S. Prehn, W. J. Toetenel (eds.), *VDM 91: Formal Software Development Methods 4th International Symposium of VDM (Noordwijkerhout, The Netherlands, October 1991) Volume 1: Conference Contributions*. Springer Verlag, 1991.

Garlan et al., 1995

D. Garlan, R. Allen, J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 12(6), pp. 17–26, 1995.

Garlan & Perry, 1995

D. Garlan, D.E. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4), pp. 269–274, April 1995.

Gatzju, 1995

S. Gatzju. *Events in an Active Object-Oriented Database System*, Doctoral Dissertation, University of Zurich, 1995.

Geiger, 1994

W. Geiger. *Qualitätslehre: Einführung, Semantik, Terminologie*. 2. Auflage, Vieweg Verlagsgesellschaft, 1994.

Genrich, 1986

H. Genrich. *Predicate/Transition Nets*. In G. Rozenberg (ed.), *Petri Nets: Central Models and Their Properties Advances in Petri Nets 1986 Part I*. Berlin (etc.): Springer Verlag, 1987.

Georgakopoulos & Hornick, 1994

D. Georgakopoulos, M. Hornick. A Framework for Enforceable Specification of Extended Transaction Models and Transactional Workflows. *International Journal of Intelligent and Cooperative Information Systems*, 3(3), pp. 599–617, September 1994.

Georgakopoulos et al., 1994

D. Georgakopoulos, M. Hornick, P. Krychniak, F. Manola. Specification and Management of Extended Transactions in a Programmable Transaction Environment. *Proceedings of the 10th International Conference on Data Engineering, (Houston, TX, 1994)*. IEEE Computer Society Press.

Georgakopoulos et al., 1995

D. Georgakopoulos, M. Hornick, A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2), pp. 119–153, April 1995.

Geppert, 1994

A. Geppert. *Methodical Construction of Database Management Systems*. Doctoral Dissertation, University of Zurich, 1994.

Geppert & Tombros, 1997

A. Geppert, D. Tombros. Logging and Post-Mortem Analysis of Workflow Executions based on Event Histories. *Proceedings 3rd International Workshop on Rules in Database Systems, (Skövde, Sweden, June 1997)*. Springer Verlag.

Geppert & Tombros, 1997a

A. Geppert, D. Tombros. Ereignisgesteuerte Workflow-Ausführung in verteilten Umgebungen. In [Jablonski et al., 1997].

Geppert et al., 1998

A. Geppert, M. Kradolfer, D. Tombros. Market-Based Workflow Management. In W. Lamersdorf, M. Merz (eds.), *Trends in Distributed Systems for Electronic Commerce, Proceedings International IFIP/GI Working Conference, TREC'98 (Hamburg, Germany, June 1998)*. Springer Verlag.

Geppert & Tombros, 1998

A. Geppert, D. Tombros. Event-Based Distributed Workflow Execution with EVE. In *Middleware 98, Proceedings IFIP International Conferences on Distributed Systems Platforms and Open Distributed Processing, (England, September 1998)*. Springer Verlag.

Gisi & Kaiser, 1991

M. Gisi, G. Kaiser. Extending A Tool Integration Language. In *Proc. First International Conference on the Software Process: Manufacturing Complex Systems, (October 1991)*. IEEE Computer Society Press, 1991.

Graw & Gruhn, 1995

G. Graw, V. Gruhn. Distributed Modeling and Distributed Execution of Business Processes.

- In W. Schaefer, P. Botella (eds.), *Proceedings European Software Engineering Conference, (Sitges, Spain, September 1995)*. Springer Verlag.
- Grefen & de Vries, 1998
P. Grefen, R. de Vries. A Reference Architecture for Workflow Management Systems. *Data and Knowledge Engineering*, 27(1), pp. 31–57, 1998.
- Griffel, 1998
F. Griffel. *Componentware: Konzepte und Techniken eines Softwareparadigmas*. Dpunkt Verlag, 1998.
- Hammer & Champy, 1994
M. Hammer, J. Champy. *Reengineering the Corporation. A Manifesto for Business Revolution*. Harper Business, 1994.
- Harel et al., 1988
D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. In *Proceedings 10th International Conference on Software Engineering, (April, 1988)*. IEEE Computer Society Press.
- Hart & Lupton, 1995
R. Hart, G. Lupton. DEC FUSE: Building a Graphical Software Development Environment from UNIX Tools. *Digital Technical Journal*, 7(2), pp. 5–19, Spring 1995.
- Hayes-Roth et al., 1995
B. Hayes-Roth, K. Pflieger, P. Lalanda, P. Morignot, M. Balabanovic. A Domain-Specific Software Architecture for Adaptive Intelligent Systems. *IEEE Transactions on Software Engineering*, 21(4), pp. 288–301, April 1995.
- Heinl & Schuster, 1996
P. Heinl, H. Schuster. Towards a Highly Scaleable Architecture of Workflow Management Systems. In *Proceedings of the 7th International Conference on Database and Expert System Applications, (Zurich, September 1996)*. Springer Verlag.
- Henninger, 1997
S. Henninger. An Evolutionary Approach to Constructing Effective Software Reuse Repositories. *ACM Transactions on Software Engineering and Methodology*, 6(2), pp. 111–140, April 1997.
- Hsu & Kleissner, 1996
M. Hsu, C. Kleissner. ObjectFlow: Towards a Distributed Process Management Infrastructure. *Distributed and Parallel Databases*, 4(2), pp. 169–, April 1996.
- Jablonski & Bussler, 1996
S. Jablonski, C. Bussler. *Workflow Management Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, 1996.
- Jablonski et al., 1997
S. Jablonski, M. Böhm, W. Schulze (eds). *Workflow-Management: Entwicklung von Anwendungen und Systemen*. Dpunkt Verlag, 1997.
- Jacobson et al., 1992
I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

Jasper, 1994

H. Jasper. Active databases for active repositories. In A. Elmagarmid and E. Neuhold (eds.), *Proceedings of the 10th International Conference on Data Engineering, (Houston, TX, February 1994)*. IEEE Computer Society Press.

Jasper et al., 1995

H. Jasper, O. Zukunft, H. Behrends. Time Issues in Advanced Workflow Management Applications of Active Databases. *Proc. of the 1st International Workshop on Active and Real-Time Database Systems (Skövde, Sweden, June 1995)*. Springer Verlag.

Jasper & Zukunft, 1997

H. Jasper, O. Zukunft. Realisierung des Verhaltensaspekts auf der Basis aktiver Datenbanksysteme. In [Jablonski et al., 1997], pp. 323–335.

Jensen et al., 1994

C.S. Jensen, J. Clifford, R. Elmasri, S.K. Gadia, P. Hayes, S. Jajodia (eds.). A consensus glossary of temporal database concepts. *SIGMOD Record*, 23(1), 1994.

Johnson & Foote, 1988

R. Johnson, B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(5), June/July 1988.

Kamath et al., 1996

M. Kamath, G. Alonso, R. Guenthoer, and C. Mohan. Providing High Availability in Very Large Workflow Management Systems. In *Advances in Database Technology— Proceedings 5th International Conference on Extending Database Technology (Avignon, France, March 25-29, 1996)*. Springer Verlag.

Kamath & Ramamrithan, 1996

M. Kamath, K. Ramamritham. Bridging the Gap between Transaction Management and Workflow Management. *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems, (Athens, GA, May 1996)*.

Kamath & Ramamrithan, 1998

M. Kamath, K. Ramamritham. Failure Handling and Coordinated Execution of Concurrent Workflows. *Proceedings 14th International Conference on Data Engineering, (Orlando, Florida, USA, 1998)*.

Kappel et al., 1995

G. Kappel, B. Pröll, S. Rausch-Schott, W. Retschitzegger. TriGS_{flow} - Active Object-Oriented Workflow Management. *Proceedings of the 28th Hawaii International Conference on System Sciences, (Hawaii, January 1995)*.

Kappel et al. 1996

G. Kappel, S. Rausch-Schott, W. Retschitzegger, M. Sakkinen. From Rules to Rule Patterns. In *Proceedings of the Conference on Advanced Information Systems Engineering, (Heraklion, Crete, May 1996)*. Springer Verlag.

Kappel et al., 1998

G. Kappel, S. Rausch-Schott, W. Retschitzegger. Coordination in workflow management systems – A rule-based approach. In W. Conen, G. Neumann (eds.), *Coordination Technology for Collaborative Applications – Organization, Processes, and Agents*. Springer Verlag, 1998.

Karbe, 1994

B. Karbe. Flexible Vorgangssteuerung mit ProMinanD. In U. Haenskamp, S. Kirn, M. Syring (eds.), *CSCW - Computer Supported Cooperative Work*. Addison-Wesley, 1994.

Kim, 1995

W. Kim. *Modern Database Systems*, Addison-Wesley, 1995.

Kirrmann et al., 1986

H. Kirrmann, T. Lalive d'Epinay, H. Stöckler. Architectures for Process Control. In R. Güth (ed.), *Computer Systems for Process Control*. Plenum Press, 1986.

Karlapalem et al., 1995

K. Karlapalem, H.P. Yeung, P.C.K Hung. CapBasED-AMS – A Framework for Capability-Based and Event-Driven Activity Management System. In *Proceedings International Conference on Cooperative Information Systems*, 1995.

Kradolfer & Geppert, 1999

M. Kradolfer, A. Geppert, and K. R. Dittrich. Workflow Specification in TRAMs. In *Proceedings 18th International Conference on Conceptual Modeling ER'99* (Paris, France, November 1999).

Krishnakumar and Sheth, 1995

N. Krishnakumar, A Sheth. Managing Heterogeneous Multi-System Tasks to Support Enterprise-Wide Operations. *Distributed and Parallel Databases*, 3(2), pp. 155–186, 1995.

Krishnamurthy & Rosenblum, 1995

B. Krishnamurthy, D. S. Rosenblum. Yeast: A General Purpose Event-Action System. *IEEE Transactions on Software Engineering*, 21(10), October 1995.

Krueger, 1992

C. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2), pp. 131–183, June 1992.

Leyman & Altenhuber, 1994

F. Leyman, W. Altenhuber. Managing Business Processes as an Information Resource. *IBM Systems Journal*, 33(2), 1994.

Leymann & Roller, 1994

F. Leymann, D. Roller. Business Process Management with FlowMark. In *Proceedings 39th IEEE Computer Conference (San Francisco, USA, 1994)*. IEEE Computer Society Press.

Luckham et al., 1995

D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4), pp. 336–355, April 1995.

Merz et al., 1994

M. Merz, K. Müller, W. Lamersdorf. Service Trading and Mediation in Distributed Computing Systems. *Proceedings of the 14th International Conference on Distributed Computing Systems (Poznan, Poland, June 1994)*. IEEE Computer Society Press.

McCarthy & Dayal, 1989

D. McCarthy, U. Dayal. The Architecture of an Active Data Base Management System. *Proceedings of the ACM SIGMOD international Conference on Management of Data (Portland, OR, June 1989)*.

McDermid & Rook, 1991

J.A. McDermid, P. Rook. Software Development Process Models. In J.A. McDermid (ed.). *Software Engineer's Reference Book*. Butterworth-Heinemann, 1991.

McLeod, 1994

R. McLeod. *Systems Analysis and Design*. Dryden Press, 1994.

Medina-Mora et al., 1992

R. Medina-Mora, T. Winograd, R. Flores, F. Flores. The Action Workflow Approach to Workflow Management Technology. In *Proceedings ACM 1992 Conference on Computer-Supported Cooperative Work (Toronto, Canada, October 31 - November 4, 1992)*. ACM Press.

Middleware Spectra, 1998

Middleware Spectra. URL <http://www.middlewarespectra.com/>, June 1998.

Miller et al., 1996

J. Miller, A. Sheth, K. Kochut, X. Wang. CORBA-based Runtime Architectures for Workflow Management Systems. *Journal of Database Management, Special Issue on Multidatabases*, 7(1), pp. 16–27, 1996.

Miller et al., 1998

J. Miller, D. Palaniswami, A. Sheth, K. Kochut, H. Singh. WebWork: METEOR's Web-based Workflow Management System. *Journal of Intelligent Information Systems*, 10(2), March 1998.

Mills, 1991

D. Mills. Internet Time Synchronization: The Network Time Protocol. *IEEE Transactions on Communication*, 39(10), pp. 1482–1493, October 1991.

Mohan et al., 1995

C. Mohan, G. Alonso, R. Günthör, M. Kamath. Exotica: A Research Perspective on Workflow Management Systems. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(1), March 1995.

Moss, 1985

E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

Moss, 1990

E. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems*, 8(2), April 1990.

Muth et al., 1998

P. Muth, D. Wodtke, J. Weissenfels, A. Kotz-Dittrich, G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems*, 10(2), pp. 159–184, 1998.

Nierstrasz & Dami, 1995

O. Nierstrasz, L. Dami. Component-Oriented Software Technology. In O. Nierstrasz, D. Tsichritzis (eds.), *Object-Oriented Software Composition*. Prentice Hall, 1995.

Oberweis et al., 1997

A. Oberweis, R. Schätzle, W. Stucky, W. Weitz, G. Zimmermann. INCOME/WF - A Petri Net Based Approach to Workflow Management. In *Tagungsband Wirtschaftsinformatik '97 (Berlin, February 1997)*.

Österle, 1996

H. Österle. Business Engineering: Von intuitiver Organization zu rationalen Workflows. In [Österle & Vogler, 1996].

Österle & Vogler, 1996

H. Österle, P. Vogler (eds.), *Praxis des Workflow Managements, Grundlagen, Vorgehen, Beispiele*. Verlag Vieweg, 1996.

OMG, 1995

The Object Management Group. *The Common Object Request Broker Architecture and Specification*. Revision 2.0, July 1995.

OMG, 1997

The Object Management Group. *CORBA Services Specification*. November 1997.

OMG, 1998a

The Object Management Group. *Notification Service, Joint Revised Submission*. January 1998.

OMG, 1998b

The Object Management Group. *Workflow Management Facility, Revised Submission*. July 1998.

Orfali et al., 1996

R. Orfali, D. Harkey, J. Edwards. *The Essential Client/Server Survival Guide*. John Wiley & Sons, 1996.

Papazoglou & Schlageter, 1998

M. Papazoglou, G. Schlageter (eds.). *Cooperative Information Systems Trends and Directions*. Academic Press, 1998.

Pintado, 1995

X. Pintado. Gluons and the Cooperation between Software Components. In O. Nierstrasz, D. Tsichritzis (eds.), *Object-Oriented Software Composition*. Prentice Hall, 1995.

Popovich, 1997

S. Popovich. *An Architecture for Extensible Workflow Process Servers*. Doctoral Dissertation, Columbia University, 1997.

Pree, 1997

W. Pree. Component-based Software Development—A New Paradigm to Software Engineering? *Software—Concepts and Tools*, 18, 1997.

Prieto-Díaz & Freeman, 1987

R. Prieto-Díaz, P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1), January 1987.

Prieto-Díaz 1991

R. Prieto-Díaz. Domain-Analysis for Reusability. In R. Prieto-Díaz, G. Arango (eds), *Domain Analysis and Software Systems Modelling*, IEEE Computer Society Press, 1991.

Purtilo, 1994

J. M. Purtilo. The POLYLITH Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1), January 1994.

Reichert & Dadam, 1998

M. Reichert, P. Dadam. ADEPT_{flex} – Supporting Dynamic Changes of Workflows Without Loosing Control. *Journal of Intelligent Information Systems*, 10(2), March 1998.

Reiss, 1995

S. Reiss. *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*. Kluwer Academic Publishers, 1995.

Reinwald, 1995

B. Reinwald. *Workflow Management in verteilten Systemen: Entwurf und Betrieb arbeitsteiliger Anwendungssysteme*. Teubner, 1995.

Riempp & Nastansky, 1997

G. Riempp, L. Nastansky. From Islands to Flexible Business Process Networks - Enabling the Interaction of Distributed Workflow Management Systems. In *Proceedings European Conference on Information Systems, (Cork, Ireland, June 1997)*.

Rusinkiewicz & Sheth, 1995

M. Rusinkiewicz, A. Sheth. Specification and Execution of Transactional Workflows. In W. Kim (ed.), *Modern Database Systems: The Object Model, Interoperability and Beyond*. Addison-Wesley, 1995.

Schäl, 1996

T. Schäl. *Workflow Management Systems for Process Organisations*. Springer Verlag, 1996.

Schulze et al., 1998

W. Schulze, C. Bussler, K. Meyer-Wegener. Standardising on Workflow-Management – The OMG Workflow Management Facility. *ACM SIGGROUP Bulletin*, 19:3, April 1998.

Schuster et al., 1996

H. Schuster, S. Jablonski, P. Heintl, C. Bussler. A General Framework for the Execution of Heterogenous Programs in Workflow Management Systems. *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (Brussels, Belgium, June 19-21, 1996)*. IEEE Computer Society Press.

Schwiderski, 1996

S. Schwiderski. *Monitoring the Behavior of Distributed Systems*. Doctoral Dissertation, University of Cambridge, 1996.

Shapiro, 1986

M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings 6th International Conference on Distributed Computing Systems (Cambridge, Massachusetts, May 19-13, 1986)*. IEEE Computer Society Press.

Shaw et al., 1995

M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

Shaw & Garlan, 1996

M. Shaw, D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

Schmidt et al., 1998

D. Schmidt, D. Levine, S. Mungee, The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21, April 1998.

Schmidt, 1999

D. Schmidt. An Architectural Overview of the ACE Framework: A Case-study of Successful Cross-platform Systems Software Reuse. *USENIX ;login*, pp. 15–25, January 1999.

Sun Microsystems, 1997

Java Beans 1.01 API Specification. <http://www.javasoft/beans/spec.html>, 1997.

Sutton et al., 1995

S. Sutton, D. Heimbigner, L.J. Osterweil. APPL/A; A Language for Software Process Programming. *ACM Transactions on Software Engineering and Methodology*, 4(3), July 1995.

Szyperski, 1997

C. Szyperski. *Component Software Beyond Object-Oriented Programming*. ACM Press/ Addison-Wesley, 1997.

Tarumi et al., 1997

H. Tarumi, K. Kida, Y. Ishiguro, K. Yoshifu, T. Asakura. WorkWeb System – Multi-Workflow Management with a Multi-Agent System. In *Proceedings ACM GROUP 97, (Phoenix, Arizona, 1997)*.

Taylor et al., 1996

R. Taylor, N. Medvidovic, K. Anderson, E. Whitehead, J. Robbins, K. Nies, P. Oreizy, D. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.

Tombros et al., 1995

D. Tombros, A. Geppert, K.R. Dittrich. Design and Implementation of Process-Oriented Environments with Brokers and Services. In B. Freitag, C. B. Jones, C. Lengauer, H.-J. Schek (eds), *Object Orientation with Parallelism and Persistence*. Kluwer Academic Publishers, 1995.

Tombros & Geppert, 1997

D. Tombros, A. Geppert. *Managing Heterogeneity in Commercially Available Workflow Management Systems: A Critical Evaluation, SWORDIES Report 4*. Department of Computer Science, University of Zurich, June 1997.

Tombros et al., 1997

D. Tombros, A. Geppert, K.R. Dittrich. Semantics of Reactive Components for Event-Driven Workflow Execution. *Proc. 9th Int'l Conf. on Advanced Information Systems Engineering (Barcelona, Spain, 1997)*. Springer Verlag.

Valetto & Kaiser, 1996

G. Valetto, G. Kaiser. Enveloping Sophisticated Tools into Computer Aided Software Engineering Environments. *Journal of Automated Software Engineering*, 3(3-4), 1996.

van der Aalst, 1998

W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

von Bültingsloewen et al., 1996

G. von Bültingsloewen, A. Koschel, R. Kramer. Active Information Delivery in a CORBA-

- based Distributed Information System. *Proceedings of the First IFCIS International Conference on Cooperative Information Systems (Brussels, Belgium, June 19-21, 1996)*. IEEE Computer Society Press.
- Wächter & Reuter, 1992
H. Wächter, A. Reuter. The ConTract Model. In [Elmagarmid, 1992]
- Wakeman & Jowett, 1993
L. Wakeman, J. Jowett. *PCTE: The Standard for Open Repositories*. New York (etc.): Prentice Hall, 1993.
- Weske et al., 1999
M. Weske, T. Goesmann, R. Holten, R. Striemer. A Reference Model for Workflow Application Development Porcess. *Proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (San Fransisco, CA, February 1999)*.
- Widom & Ceri, 1996
J. Widom, S. Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kauffman, 1996.
- WfMC, 1994
Workflow Management Coalition. *The Workflow Reference Model*. Document No. TC00-1003, November 1994.
- WfMC, 1995
Workflow Management Coalition. *Workflow Process Definition Read Write Interface*. Document No. WfMC-WG01-1000, February 1995.
- WfMC, 1996a
Workflow Management Coalition. *Workflow Client Application (Interface 2) Application Programming Interface (WAPI) Specification*. Document No. WfMC-TC-1009, October 1996.
- WfMC, 1996b
Workflow Management Coalition. *Workflow Standard - Interoperability Abstract Specification*. Document No. WfMC-TC-1012, October 1996.
- WfMC, 1996c
Workflow Management Coalition. *Workflow Standard - Interoperability Internet e-mail MIME Binding*. Document No. WfMC-TC-1018, October 1996.
- WfMC, 1996d
Workflow Management Coalition. *Audit Data Specification*. Document No. WfMC-TC-1015, November 1996.
- WfMC, 1998
Workflow Management Coalition. *Interface 1: Process Definition Interchange – Process Model*. Document Number WfMC TC-1016-P, 7.05 beta, August 1998.
- Wodtke, 1997
D. Wodtke. *Modellbildung und Architektur von verteilten Workflow-Management-Systemen*. Dissertation. Infix, 1997.
- Worah & Sheth, 1997
D. Worah, A. Sheth. Transactions in Transactional Workflows. In S. Jajodia, L. Kerschberg (eds.), *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.

Curriculum Vitae

Last name: Tombros

First name: Dimitrios

Date of birth: 6. September 1965

Place of birth: Athens, Greece

1971 - 1976 Primary school in Athens, Greece

1977 - 1983 Gymnasium in Athens, Greece (Athens College)

1985 - 1991 University of Zurich, Switzerland
Studies in Business and Information Technology. Graduation in June 1991

1991 - 1992 Junior developer at Siemens Nixdorf Informationssysteme, Munich, Germany

1992 - 1994 Stanford University, California, USA
Master of Science in Computer Science. Graduation in June 1994

1994 - 1999 Research assistant at the Department of Computer Science,
University of Zurich, Switzerland

1999 PhD Thesis “An Event- and Repository-Based Component Framework
for Workflow System Architecture”

